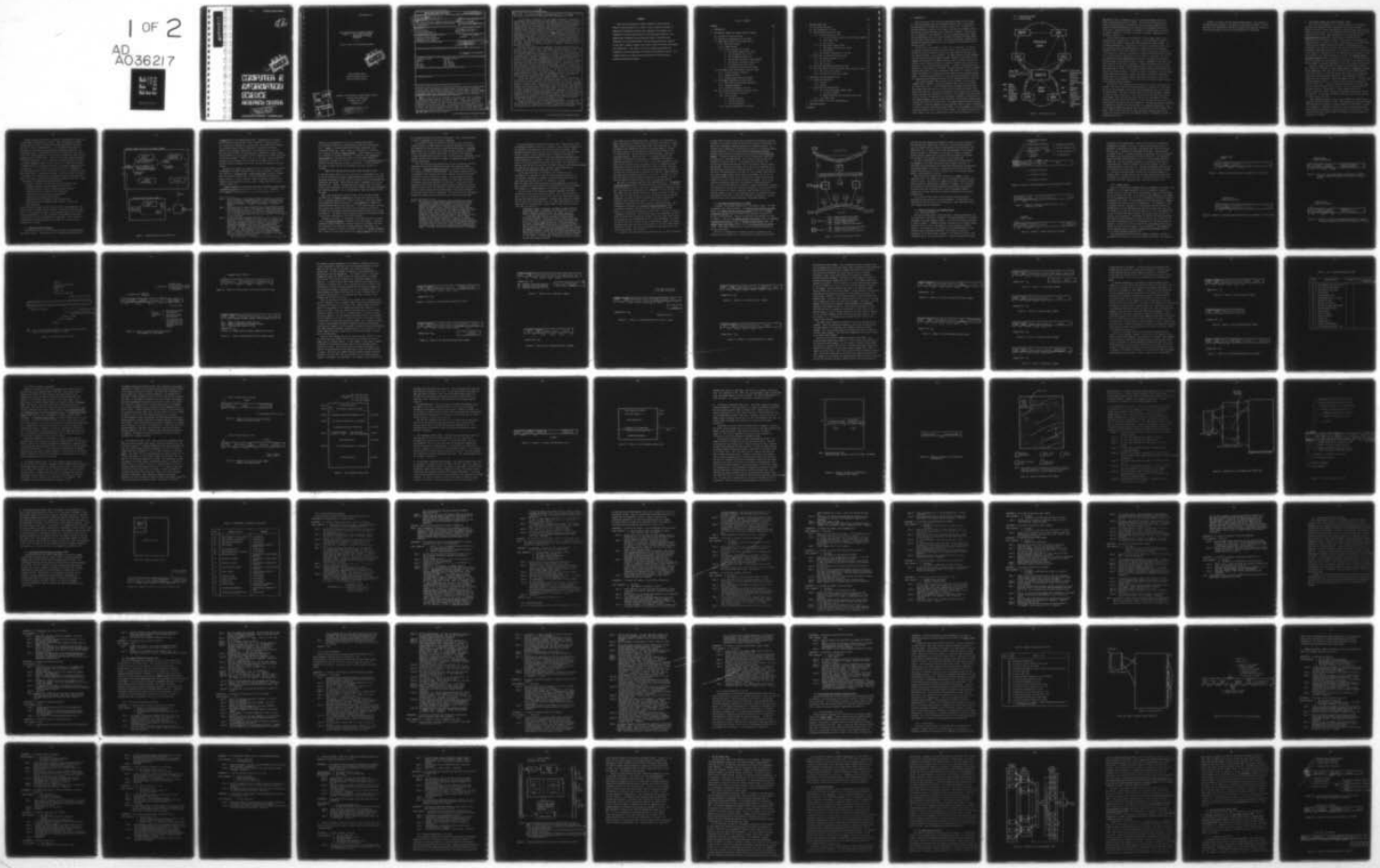
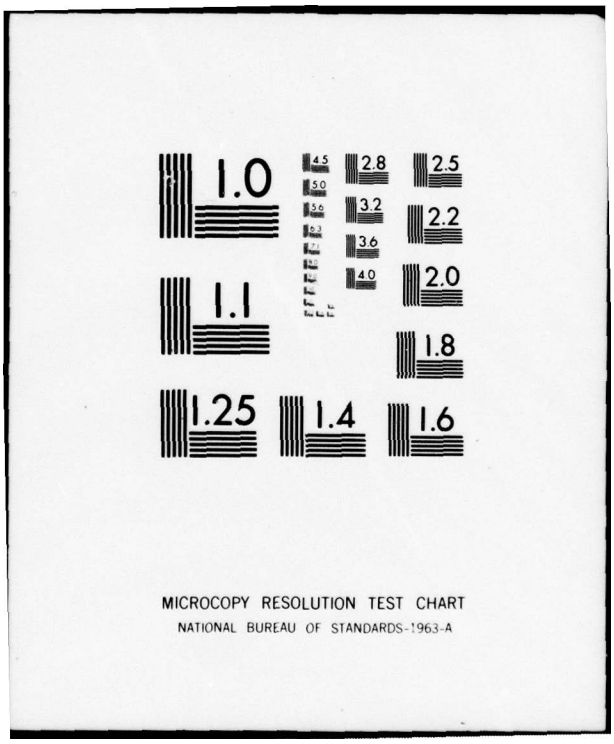
AD-A036 217 OHIO STATE UNIV COLUMBUS COMPUTER AND INFORMATION SC--ETC F/G 9/2
THE ARCHITECTURE OF A DATABASE COMPUTER. PART III. THE DESIGN O--ETC(U)
DEC 76 D K HSIAO, K KANNAN N00014-75-C-0573
UNCLASSIFIED OSU-CISRC-TR-76-3 NL

1 OF 2
AD
A036217




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

TECHNICAL REPORT SERIES

12

COMPUTER & INFORMATION SCIENCE RESEARCH CENTER

THE OHIO STATE UNIVERSITY COLUMBUS, OHIO

(OSU-CISRC-76-3)

THE ARCHITECTURE OF A DATABASE COMPUTER PART III: THE DESIGN OF THE MASS MEMORY AND ITS RELATED COMPONENTS

by

David K. Hsiao and Krishnamurthi Kannan

Work performed under
Contract N00014-75-C-0573
Office of Naval Research

Computer and Information Science Research Center
The Ohio State University
Columbus, Ohio 43210

December 1976

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: OSU-CISRC-TR-76-3	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): "The Architecture of a Database Computer, Part III: The Design of the Mass Memory and its Related Components"	5. TYPE OF REPORT & PERIOD COVERED: Technical Report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): David K. Hsiao, Krishnamurthi Kannan	8. CONTRACT OR GRANT NUMBER(s): N00014-75-C-0573
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Office of Naval Research, Information Systems Program, Washington, D.C. 20360	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 4115-A1
11. CONTROLLING OFFICE NAME AND ADDRESS	12. REPORT DATE: December, 1976
	13. NUMBER OF PAGES: 141
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report)
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report): Approved for public release; distribution unlimited.

Scientific Officer	DDC New York Area
ONR BRO	ONR 437
ACO	ONR, Boston
NRL 2627	ONR, Chicago
ONR 102IP	ONR, Pasadena

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Database computer; security; clustering, partitioned content addressable memory; security atom; name mapping; structure memory; microprocessor; functional specialization; keyword transformation unit; structure memory; structure memory information processor; index transformation unit; mass memory; database command and control unit; track information processor; security filter processor.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This is the last of the three-part series which deals with the design of a back-end computer known as the database computer (DBC). The concepts and capabilities of the DBC were presented in Part I. Schematically, the DBC architecture consists of two loops of memories and processors, namely, the structure loop and the data loop. The structure loop is composed of four components: the structure transformation unit (KXU), the structure memory (SM), the structure memory information processor (SMIP) and the index translation

OVER

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

unit (IXU). The design philosophy, implementation details and hardware organizations of the structure loop components were documented in Part II.[2]

In this report, the design of the data loop is presented. In addition, the database command and control processor (DBCCP), which regulates the operations of both the structure and data loops and interfaces with the front-end computer systems, is also presented. The DBCCP processes all DBC commands received from the front-end computer systems, schedules the execution of the commands on the basis of the command type and priority, enforces security on a selective basis, clusters records to be stored in the DBC and routes the response set to the front-end computer systems. A number of table memories and processors is incorporated in the DBCCP. The major memories include the command argument table memory, the file information table memory, the security information table memory, the command status table memory and the database response memory. The main processors consist of the structure-loop-interface processor, the command-check-and-response processor and the command-translation processor. Although the design of the DBCCP is straightforward, the details are rather involved. To this end, we attempt to provide a comprehensive presentation in Section 2.

The data loop consists of two components, the mass memory (MM) and the security filter processor (SFP). The design of the MM (presented in Section 3) is based on the concept of partitioned content-addressable memory (PCAM). In this PCAM implementation, a partition is a cylinder of a moving-head disk unit. The cylinder is made content-addressable by incorporating track information processors (TIPs) (one for each track of a cylinder) for concurrent processing of the tracks of a cylinder. Furthermore, the disk read/write mechanism is modified to allow parallel read/write of all the tracks of a cylinder. The choice of a processor-oriented implementation using TIPs vs. a memory-oriented implementation using a large cylinder buffer is argued. Management of MM orders, and their execution by the TIPs are discussed. Garbage collection and space reclamation are also discussed in considerable details in terms of compaction and update operations of the MM. By far the most powerful operation of the MM is the search and retrieve operation. The MM is capable of searching for and retrieving records which satisfy queries. Because the records in the MM are addressed by content and carry no address pointers, they need no updating as long as the records exist in the database. This is true even if the security specifications of the database change frequently.

The security filter processor provides the type B security enforcement and sorting. The type B security enforcement mechanism is provided for those users who do not take advantage of the type A security mechanism based on the concept of security atoms. The type A security incurs less security overhead. However, it needs the user's cooperation. First, the user must understand the security atom concepts; then, the user must convey the security requirements in terms security attributes of his data records. On the other hand, the type B security mechanism does not require such user cooperation. Nevertheless, posterior checking of response data against full file sanctions is an expensive undertaking. The sort mechanism enables the response data to be ordered by values of certain attributes. This is usually the way that the user application programs would like to receive the records in the front-end computer systems. The design of the SFP is presented in Section 4.

Finally, in Section 5 we have some concluding remarks. The conclusion of the DBC design has prompted us to undertake a new series of studies. In the new series, the feasibility of the DBC in supporting hierarchical, network and relational data models and their related systems will be presented.

PREFACE

This work was supported by Contract N00014-75-C-0573 from the Office of Naval Research to Dr. David K. Hsiao, Associate Professor of Computer and Information Science, and conducted at the Computer and Information Science Research Center of The Ohio State University. The Computer and Information Science Research Center of The Ohio State University is an interdisciplinary research organization which consists of the staff, graduate students, and faculty of many University departments and laboratories. This report is based on research accomplished in cooperation with the Department of Computer and Information Science. The research contract was administered and monitored by The Ohio State University Research Foundation.

TABLE OF CONTENTS

1. INTRODUCTION

This is the last of the three-part series which deals with the design of a back-end computer known as the database computer (DBC). The concepts and capabilities of the DBC were presented in Part I [1]. Schematically, the DBC architecture consists of two loops of memories and processors, namely the structure loop and the data loop as depicted in Figure 1. The structure loop is composed of four components: the keyword transformation unit (KXU), the structure memory (SM), the structure memory information processor (SMIP) and the index translation unit (IXU). The design philosophy, implementation details and hardware organizations of the structure loop components were documented in Part II [8].

In this report, the design of the data loop is presented. In addition, the database command and control processor (DBCCP), which regulates the operations of both the structure and data loops and interfaces with the front-end computer systems, is also presented. The DBCCP processes all DBC commands received from the front-end computer systems, schedules the execution of the commands on the basis of the command type and priority, enforces security on a selective basis, clusters records to be stored in the DBC, and routes the response set to the front-end computer systems. A number of table memories and processors is incorporated in the DBCCP. The major memories include the command argument table memory, the file information table memory, the security information table memory, the command status table memory and the database response memory. The main processors consist of the structure-loop-interface processor, the command-check-and-response processor and the command-translation processor.

The command argument table contains actual parameters of the incoming commands such as queries, clustering conditions and records for insertion. For each active file, a file information table maintains information about mass memory space allocation to the file and certain security related information about the file. The security information table contains file sanctions, atomic privilege lists and security descriptors. The command status table keeps track of the state of execution of the outstanding commands, the whereabouts of the command arguments, and the command priorities. The database response memory contains the output from the data loop. The command-check-and-response processor is capable of receiving commands from the front-end computer systems, responding to their interrupts, performing security checks on certain commands, and forwarding authorized response data to the front-end systems. The command translation processor converts each access command sent by the front-end systems into a set of

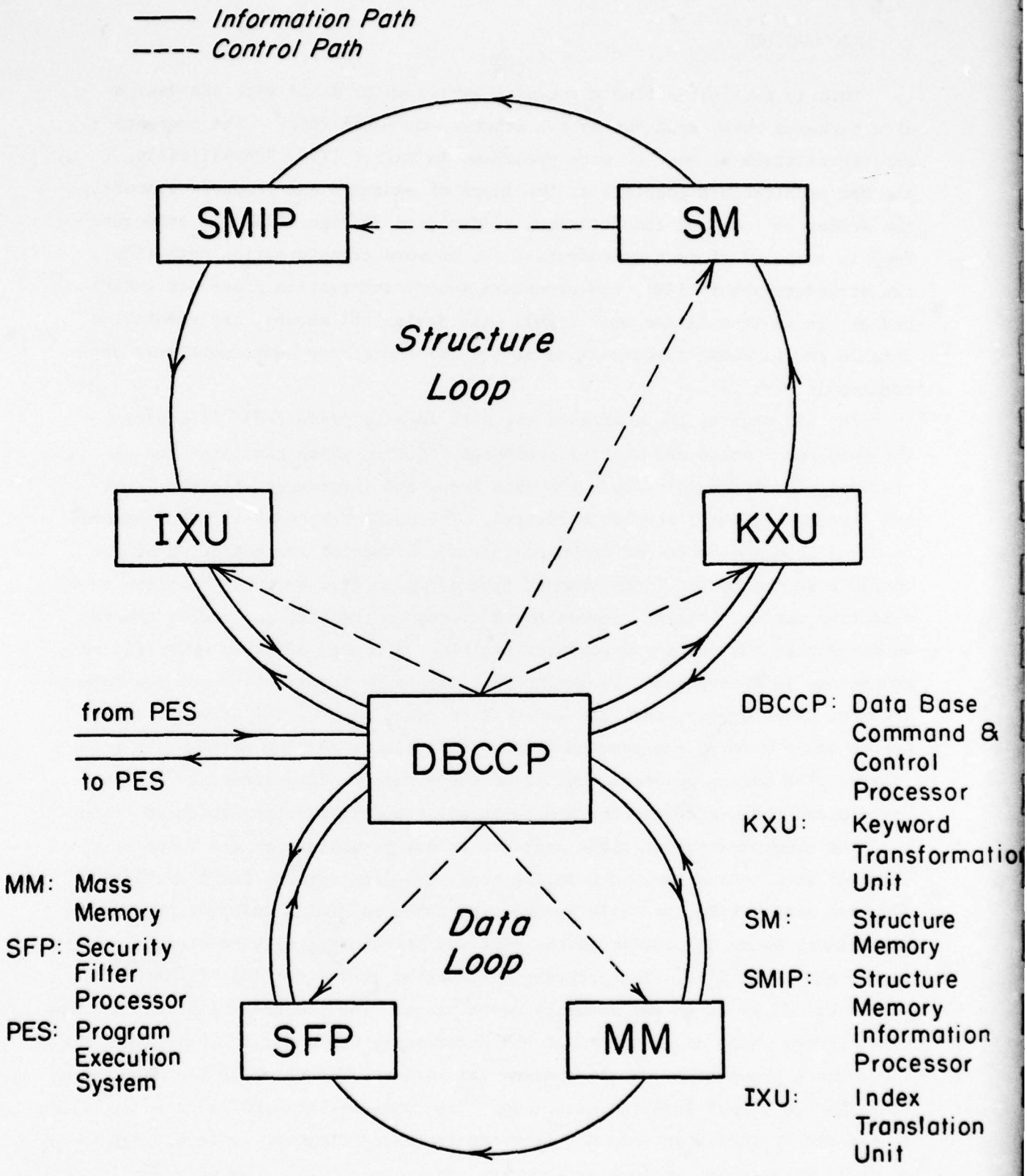


Figure 1. Architecture of DBC

mass memory orders for subsequent access to content-addressable partitions known as the minimal access units (MAUs). The structure-loop-interface processor initiates requests for information from the structure memory and responds to interrupts generated by the structure loop components (principally by the IXU). Although the design of the DBCCP is straightforward, the details are rather involved. To this end, we attempt to provide a comprehensive presentation in Section 2.

The data loop consists of two components, the mass memory (MM) and the security filter processor (SFP). The design of the MM (presented in Section 3) is based on the concept of partitioned content-addressable memory (PCAM). In this PCAM implementation, a partition is a cylinder of a moving-head disk unit. The cylinder is made content-addressable by incorporating track information processors (TIPs) (one for each track of a cylinder) for concurrent processing of the tracks of a cylinder. Furthermore, the disk read/write mechanism is modified to allow parallel read/write of all the tracks of a cylinder. The choice of a processor-oriented implementation using TIPs vs. a memory-oriented implementation using a large cylinder buffer is argued. Management of MM orders, and their execution by the TIPs are discussed. Garbage collection and space reclamation are also discussed in considerable details in terms of compaction and update operations of the MM. By far the most powerful operation of the MM is the search and retrieve operation. The MM is capable of searching for and retrieving records which satisfy queries. Because the records in the MM are addressed by contents and carry no address pointers, they need no updating as long as the records exist in the database. This is true even if the security specifications of the database change frequently.

The security filter processor provides the type B security enforcement and sorting. The type B security enforcement mechanism is provided for those users who do not take advantage of the type A security mechanism based on the concept of security atoms. The type A security incurs less security overhead. However, it needs the user's cooperation. First, the user must understand the security atom concept; then, the user must convey the security requirements in terms security attributes of his data records. On the other hand, the type B security mechanism does not require such user cooperation. Nevertheless, posterior checking of response data against full file sanctions is an expensive undertaking. The sort mechanism enables the response data to be ordered by values of certain attributes. This is usually the way the user application programs would like to receive the records in the front-end computer systems. The design of the SFP is presented in Section 4.

Finally, in Section 5 we have some concluding remarks. The conclusion of the DBC design has prompted us to undertake a new series of studies. In the new series, the feasibility of the DBC in supporting hierarchical, network and relational data models and their related systems will be presented.

2. THE DATABASE COMMAND AND CONTROL PROCESSOR (DBCCP)

Since the DBC is designed to be a back-end computer in an integrated information and computing system consisting of one or more front-end computers, the user of such a system does not directly interact with the DBC. Instead, the DBC receives all its commands from the front-end computers which interface with the users. These systems are collectively known as the program execution system (PES). The existence of the PES has some implications for the design of the DBC. In particular, we can assume that commands received by the DBC do not have syntactic errors. We can also demand certain rigorous commands formats from the PES; such demands would be unreasonable if the DBC had to interact with users directly. Nevertheless, considerable bookkeeping chores have to be performed by the DBC. These chores relate primarily to keeping track of the commands and their arguments as they are processed by the various components of the DBC. The database command and control processor (DBCCP) is responsible for carrying out these and other chores in a manner that will maximize the utilization of the components of the DBC and ensure the response requirements as set by the database administrator.

Basically, commands received from the PES (in predetermined formats), are recognized by the DBCCP as either access commands or preparatory commands. Access commands are those that require the DBC to access the mass memory; preparatory commands are those that precede and follow access commands and convey important housekeeping information. Access commands are further divided into three categories - those that undergo the type A security check, those that undergo the type B security check and those that undergo no security check. Type A security checks use the concept of security atoms [1,2,3] to enforce security; type B security checks use file sanctions [2] to enforce security. The category of access commands which do not require any security checks, is used for loading the database with records of a new file. In this case, the DBCCP merely makes sure that the user has the right to create the file.

After undergoing security checks, access commands are translated into orders that can be processed by the MM. During translation, an access command involving insertion activates a clustering mechanism in the DBCCP. A record to be inserted is "clustered" according to a set of clustering predicates (called clustering conditions in [2]) specified by the file creator. These clustering predicates enable the DBCCP to determine the MAU into which the record must be inserted.

Records retrieved by the MM, as a result of the execution of orders sent by the DBCCP, are transmitted to the SFP. Here, the records may undergo security checks if type B protection has been specified for the user on the file to which the records belong. It is important to note here the difference in processing for the two types of protection mechanisms. In specifying the type A protection for a user of a given file, the creator of the file tacitly assumes that the file can be protected in terms of record aggregates called security atoms. [We shall give a more rigorous definition of security atoms later]. Such a specification enables the DBCCP to check for security before the access is made. The type B protection mechanism is based on full file sanctions and can only be carried out after the access is completed. Type B protection is enforced by the SFP. Depending on user requirements, a file creator may specify the type A protection for one user and the type B protection for another. The SFP can also be instructed by the DBCCP to sort records that are retrieved by the MM. The sorting is done on the basis of values of some attributes specified by the user. Figure 2 illustrates the command 'paths' discussed above.

In summary, the DBCCP is charged with the following functions:

- Accepting commands and their arguments from the PES.
- Performing security checks on commands whenever possible.
- Clustering records according to user requirements.
- Interfacing with the structure loop components.
- Translating PES commands into MM orders.
- Interfacing with the security filter processor (SFP).
- Maintaining various information tables needed to perform the above functions.

In the following sections, we first discuss some of the subtleties of the DBC data model. Although the main concepts and facilities were discussed in Part I, it is necessary to expand on the discussion here in order to gain a fuller understanding of the data structures and algorithms of the DBCCP. We then propose a physical organization of the DBCCP. This is followed by a discussion of the data structures in the DBCCP and the logic of the components of the DBCCP.

2.1 The DBC Data Model Revisited

As mentioned in Part I, the DBC directly supports the attribute-based data model [1,3,4]. An important unit of information within the DBC is

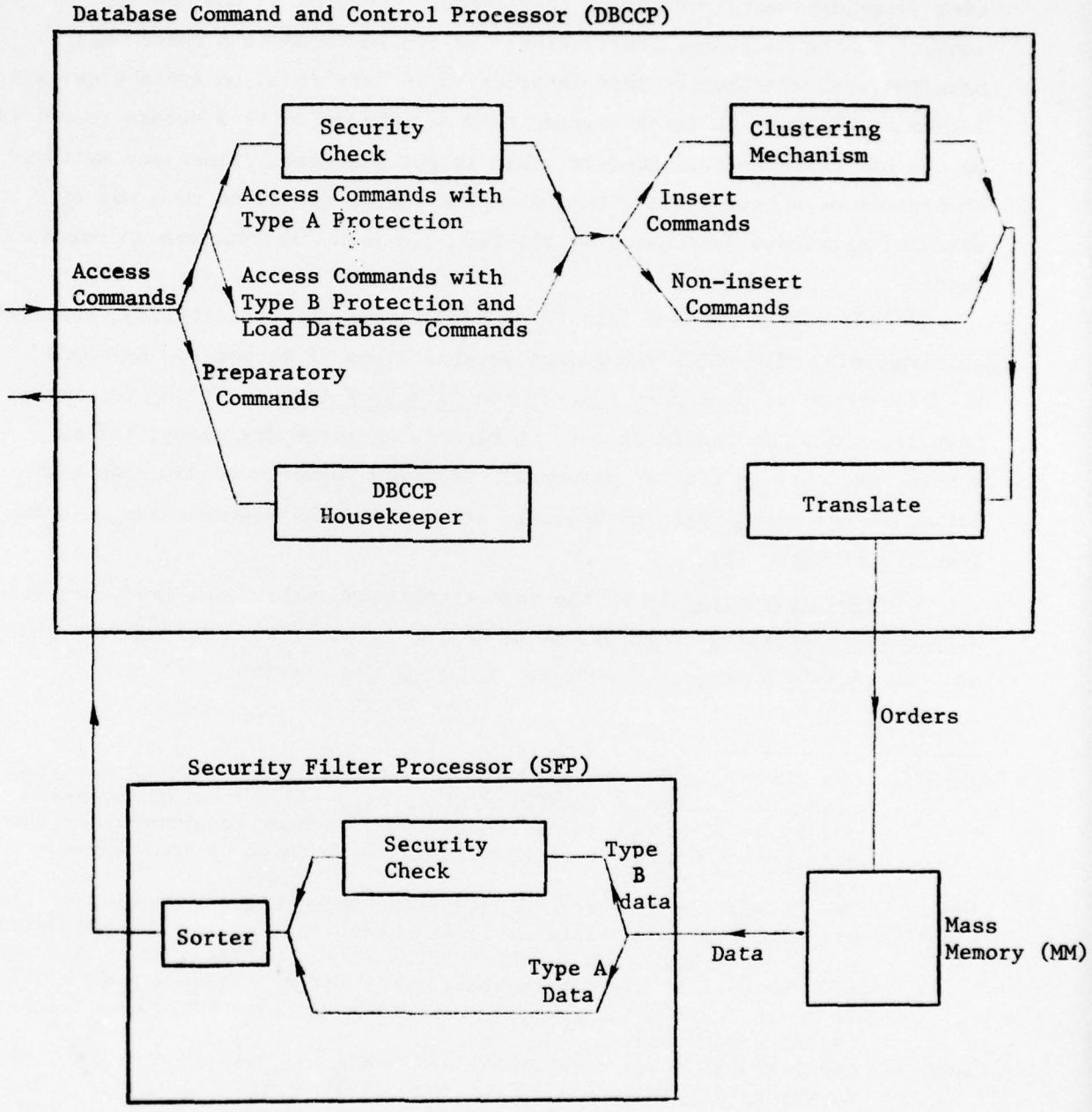


Figure 2. Command Execution in the Data Loop

a keyword which is an attribute-value pair.[1] Information can be stored into or retrieved from the DBC in terms of records; a record is made up of a collection of keywords and a record body. The record body is composed of a (possibly empty) string of characters.[2] Records in the DBC are subject to the following restriction: No two keywords in a record may have the same attribute. This restriction is introduced to avoid a certain logical problem.[3] At first glance, this may appear to be a severe restriction on the use of keywords. However, this is not the case. Since any multiple occurrence of an attribute within a record can be mapped on to a set of distinct attribute surrogates by the PES, the user can continue to use such records.

Records may be grouped into files for reasons of accessibility, security and ownership. The DBCCP recognizes several types of keywords. Keywords are classified as directory type or non-directory type according to whether they are stored in the SM or not. Directory keywords are classified as simple, security or cluster keywords. Security keywords are used in the formation and recognition of security atoms. Cluster keywords are used to identify clusters [2].

A keyword predicate is of the form <attribute, relational operator, value>. A relational operator could be one of (=, $\neq$, $\geq$, >, $\leq$, <). A keyword K is said to satisfy a keyword predicate T if a) the attribute of K is

---

Note 1. The definition of an attribute is largely intuitive. It can mean a class, a quality, a characteristic, etc. In the attribute-based model, an attribute in left undefined in order to encompass a wide spectrum of meanings. A keyword is represented as (Attribute = value).

Note 2. The definition of a record given here is a slight extension of the definition given in Part I. This extension concerns the inclusion of a record body in a record. In practice, a file creator may not wish to specify his entire record as a set of keywords, especially if he is certain that he will not query the file based on the contents of the record body.

Note 3. Consider the query ((PART=NAIL)$\wedge$(PART$\neq$NAIL)). If a record has two keywords PART=NAIL, PART=BOLT, then this record will satisfy (see definition of a record satisfying a query later in the text) the above query although the query is illogical. This problem is eliminated by redefining the record as containing keywords PART1=NAIL, PART2=BOLT. Then, the above query will not retrieve the record. It can be shown that the record will satisfy no illogical query, while enabling a user to specify any logical query. It should be noted that in the above examples although PART1 and PART2 are distinct attributes the domains of the values of PART1 and PART2 may be identical.

This problem was first brought to our attention by Ron Fagin [7].

identical to the attribute in T and b) the relation specified in the relational operator of T holds between the value of K and the value in T. A query is a boolean expression of keyword predicates in the disjunctive normal form. Thus, a query is a disjunction of conjuncts. Such conjuncts are known as query conjuncts. A query conjunct, of course, is a conjunction of keyword predicates. A record in a file satisfies a query conjunct, if each and every predicate in the query conjunct is satisfied by a keyword in the record. A record in a file satisfies a query if it satisfies at least one query conjunct in the query. To give an example of the types of queries that may be recognized by the DBC, consider the following:

((DEPT='TOY')∧(SALARY<10,000))∨((DEPT='BOOK')∧(SALARY>50,000))

If the above query refers to a file of employees of a department store, then it will be satisfied by records of those employees working either in the toy department and making less than 10,000, or working in the book department and making more than 50,000. Notice that the query is meaningful only for the specified file. Queries, as defined above, are used not only to retrieve records from the database, but are also used to specify protection requirements and clusering conditions. Let us now discuss how these are achieved in the DBC model.

When a user declares his intent to access a file, the PES provides the DBCCP with the database capability of the user with respect to that file. A database capability is a couple of the form <(file name, default access descriptor),file sanction set>. A file sanction is a conjunct of keyword predicates and an access descriptor. A file sanction merely specifies that records which satisfy the conjunct can be accessed by the user in accordance with the access descriptor which encodes the types of access that are permitted. The default access descriptor specifies the accesses allowed on records not satisfying any of the file sanctions.

Consider now the type A protection. At the time the file is created, a set of security descriptors is provided by the file creator. A security descriptor generally specifies a range of values of an attribute such that all the values in the range have the same protection requirement. An attribute occuring in a security descriptor is known as a security attribute and a security attribute and value pair is a security keyword if the value is

in the range specified in the security descriptor. Such a security keyword is said to be derived from the security descriptor.

As records of a file are loaded into the DBC as a part of the file creation process, the DBCCP extracts the security keywords of the records and determines the security atoms to which the records belong. A security atom defines a set of records each of whose security keywords is derivable from a unique set of security descriptors. By the end of the creation process, the DBCCP would have built a set of security atoms. Furthermore, no two security atoms can have a record in common - an important property of the security atom concept.

For a user who has been provided with the type A protection, the file sanctions of his database capabilities contain only security keyword predicates. [A security keyword predicate is a predicate whose attribute is a security attribute as defined above]. When the user accesses the file, the associated file sanctions are used to determine the access privileges of the user on each of the security atoms. An access query can then be accepted (or rejected) depending on whether the access type is permitted (or denied) on the atoms referred to by the query. We note that a file sanction can affect more than one security atom and a security atom can be affected by more than one file sanction. When more than one file sanction affects a security atom, then the intersection of the access descriptors of all such file sanctions is considered. The result of such an intersection defines the (atomic) access privilege (list) of the user on that atom.[1]

Note 1. The notion of an atom defined in terms of minterms of keywords for grouping of records having common keywords is due to Wong and Chiang [5]. McCauley [3] applied it to describe security sensitive information and called such atoms security atoms. However, McCauley's security atoms had severe limitations when used in real-world applications. The number of atoms could easily grow very large. The reason for this lies in his method of defining security atoms. Each security atom was defined by a unique set of security keywords occuring in the records of a file. We shall now show how such a definition can cause an atoms-explosion (not an atomic explosion!). Suppose, for example, a database creator specifies that all records which have a security attribute, called salary, with values between 10,000 and 100,000 are to be protected in a certain way. Then, if salary were the only security attribute, then we could potentially have 90,000 security atoms. It is easy to see that if there are two such (continued)

Let us recapitulate all that we have said so far. There are four pieces of information related to the type A security. First, security descriptors are provided by the creator of a file before the file is created. These descriptors specify what to look for in the records as they are loaded into the DBC. Second, a list of security atoms is created by the DBCCP by determining the unique subsets of the security descriptors satisfied by the security keywords of the records. Third, when a file is to be accessed by a user, his file sanctions are made known to the DBCCP. By comparing the security atoms and the file sanctions, the DBCCP creates an atomic access privilege list AAPL for the user of the file. The AAPL is the fourth piece of security related information maintained by the DBCCP.

In the case of the type B protection, file sanctions are not constrained to be specified in terms of security keyword predicates. It is, therefore, meaningless to attempt to build atomic privilege lists in cases where the user has the type B protection. Without AAPL, the DBCCP cannot carry out security checks before the accesses are made (A request for record insertion is an exception to the above statement. This exception is, however, handled by the security filter processor).

Let us now turn to the clustering strategies provided by the DBCCP. Clustering is done on two levels in the DBCCP. First, the allocation of MAUs to files is carried out in such a way, that no two files share an MAU. How does this scheme help performance? To answer this question, we observe that if files were to share MAUs, then it is likely that records of a file would tend to be dispersed over a larger number of MAUs than would be the

---

(continued) security attributes, the number of atoms would be astronomical indeed. In general, the maximum number of security atoms in McCauley's system would be ΠN_i where N_i is the number of security keywords of the i-th security attribute.

In the DBC, the concept of security descriptor has been explicitly introduced to limit the number of security atoms to reasonable levels. The database creator can now define atoms to be a collection of records which have certain attributes whose values are within a range indicated by a security descriptor. He specifies a particular keyword value in a security descriptor only if he needs to protect a record with that keyword value in a way different from records with other neighborhood values. Consider, the example used above. The security descriptor would specify the range $10,000 \leq \text{Salary} \leq 90,000$, and the system would construct only one security atom if this is the only security sensitive information in the records. In general, the maximum number of security atoms is $\Pi N'_i$ where N'_i is the number of security descriptors specified for the i-th security attribute.

case if MAUs were not shared among files. Therefore, it is reasonable to expect that queries for record retrieval would result in access to a larger number of MAUs if MAUs are shared among files than if they are not. Since performance would undoubtedly improve if the average number of accesses per query is kept low, we conclude that allocating entire MAUs to a file is a sound decision. We may consider the above strategy, a file-level clustering strategy. The second level of clustering is based on the principle of enlisting the cooperation of the database file creator to determine the position of a record within a set of MAUs allocated to a file. Since the file creator is not, in general, aware of the addresses of MAUs allocated to his file, the DBCCP allows the creator to specify conditions which may be satisfied by one or more records already existing in an MAU, in order that a new record can be inserted into that MAU. These conditions have been called clustering conditions in Part I [2]. A creator of a file may elect not to specify clustering conditions for his records. In this case, the DBCCP will assign records to MAUs in an arbitrary manner.

Two types of clustering conditions were identified in [2]. The mandatory clustering condition (MCC) is a query which must be satisfied by one or more records existing in an MAU in order that a new record may be inserted in that MAU. Each record for insertion may be accompanied by at most one MCC. The PES usually uses the same MCC with each of a group of records in order to ensure that all member records of the group are inserted into the same MAU. Frequently, more than one MAU may each have one or more records which satisfy the MCC accompanying a record. In such cases, we need a mechanism to choose one of the MAUs in which to insert the record. Such a mechanism is provided by the optional clustering conditions (OCCs). An optional clustering condition is also a query similar to the MCC. However, a record may be accompanied by several OCCs. With each of the OCCs, a weightage is associated. The insertion process then determines the MAU in which the record is to be placed as follows: The set of MAUs each of which has at least one record satisfying the MCC is first determined. For each of the MAUs in this set, a cluster weight is calculated by summing the weights associated with those OCCs that are satisfied by one or more records already existing in the MAU. The record to be inserted is then placed in the MAU whose cluster weight is the greatest.

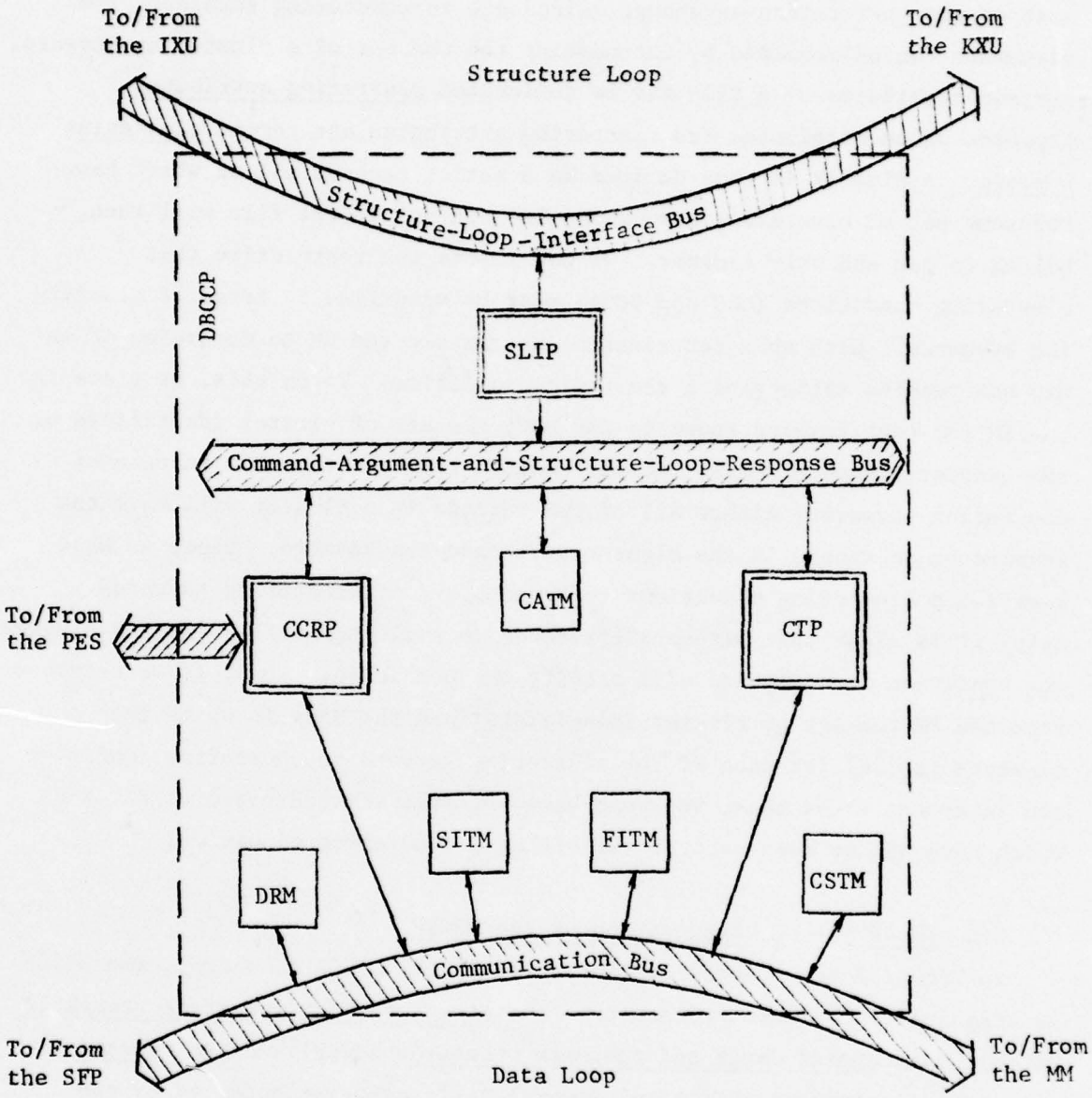
In order for the above clustering mechanism to work, we need to determine

if there exists records in an MAU which satisfies the clustering conditions. Obviously, if we need to access the MAU for this prupose, we would have lost most of the performance advantage gained due to clustering records. The situation can be remedied by introducing the concept of a clustering keyword. Certain attributes of a file may be designated clustering attributes. Keywords whose attributes are clustering attributes are termed clustering keyword. A cluster is then defined as a set of records all of which have the same set of clustering keywords. Each record in the file will then belong to one and only cluster. We now impose the restriction that clustering conditions (MCC and OCCs) must be specified in terms of clustering keywords. With this restriction, we can use the SM to determime if an MAU has records satisfying a clustering condition. To do this, we store in the SM for each keyword known to the DBC, the set of cluster identifiers of the clusters, some of whose records contain the keyword. Now in case of clustering keywords, either all of the records in a cluster will have the keyword or no record in the cluster will have the keyword. Since we have restricted clustering conditions to be composed of clustering keywords only, it is clear that either all records in a cluster will satisfy a clustering condition or no record will satisfy the condition. Thus, if we retrieve from the SM the set of cluster identifiers (and the MAUs in which the clusters reside) for each of the clustering keyword in clustering condition and intersect these sets, we would have obtained the addresses of the MAUs which have one or more records satisfying a clustering condition.

2.2 The Physical Organization of the DBCCP

In Figure 3 the physical organization of the DBCCP in shown. The DBCCP is organized into three processors - the structure loop interface processor (SLIP), the command check and response processor (CCRP) and the command translation processor (CTP); and several table memories accessed by one or more of the processors. Some of the important table memories are shown in Figure 3. These are the command argument table memory (CATM), the security information table memory (SITM), the database response memory (DRM), the command status table memory (CSTM), and the file information table memory (FITM). These tables contain most of the data structures manipulated by the three processors.

The SLIP is responsible for interfacing with the components of the structure loop (see Figure 1). It accepts service requests from the CCRP



Memories
- CATM: Command Argument Table Memory
- CSTM: Command Status Table Memory
- FITM: File Information Table Memory
- DRM: Database Response Memory
- SITM: Security Information Table Memory

Processors
- CCRP: Command Check and Response Processor
- CTP: Command Translation Processor
- SLIP: Structure Loop Interface Processor

Figure 3. Physical Organization of DBCCP

and CTP, and issues appropriate commands to three of the four components of the structure loop, namely, KXU, SM and IXU. The data received from the loop as a result of the execution of these commands, are passed back to the CCRP or the CTP. In order to ensure that the components of the structure loop are utilized to the maximum, the SLIP must be capable of handling multiple service requests and of matching service requests with response data as soon as they are made available by the structure loop.

The CCRP is responsible for receiving commands from the PES, placing the arguments in the CATM, performing security checks if the type A protection has been specified, and transmitting response data from the database response memory to the PES. In applying security checks the CCRP requests the services of the SLIP. Response data in the database response memory is made available by the SFP.

When the CCRP has completed the processing of a command, the CTP translates each of the command into a set of MM orders and transmits these orders to the MM. The status of a command in the CSTM is used to indicate when the CCRP has completed the processing of a command. It is important to note that the three processors operate asynchronously and communicate with each other only via status table. Such an arrangement makes it possible for the three processors to operate concurrently at the maximum possible rate.

In the sections that follow we first discuss the data structures maintained by the processors in the table memories. We then present the algorithms carried out by each of the processors. Data structures local to the processors are also described.

2.3 DBCCP Data Structures and Command Formats

2.3.1 Basic Data Formats

The basic building block of query conjuncts, clustering conditions, and file sanctions is the keyword predicate. A keyword predicate is of the form <attribute, relational operator, value>. The format of a keyword predicate is shown in Figure 4. Keywords occuring in records also use the same format, with bits 3-7 set to zero. The format of a query conjunct is shown in Figure 5. This format is used in queries for retrieval/deletion, in file sanctions and in clustering conditions. The format of a record as transmitted by the PES is shown in Figure 6. The format of a query in the disjunctive normal form and the format of a clustering condition are shown

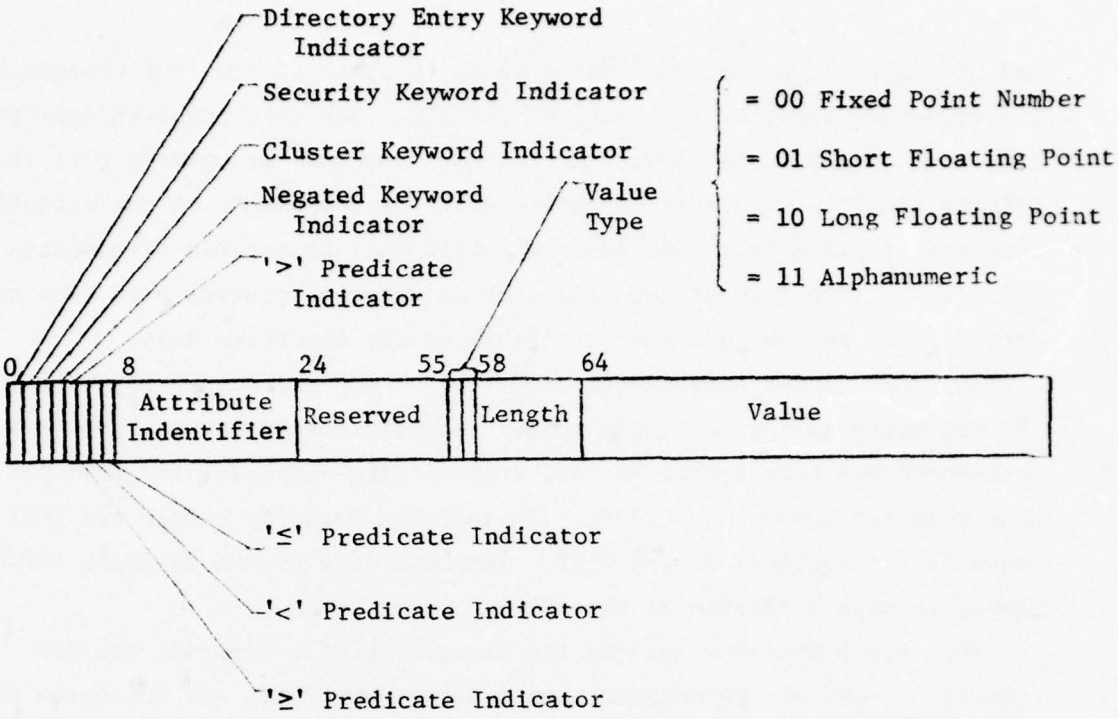


Figure 4. Format of a Keyword Predicate (T) as Received by the DBCCP

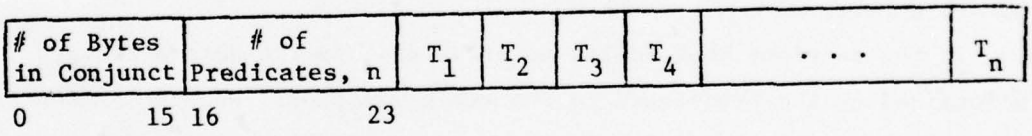


Figure 5. Format of a Predicate Conjunct ($T_1 \wedge T_2 \wedge \ldots \wedge T_n$) as Received by the DBCCP

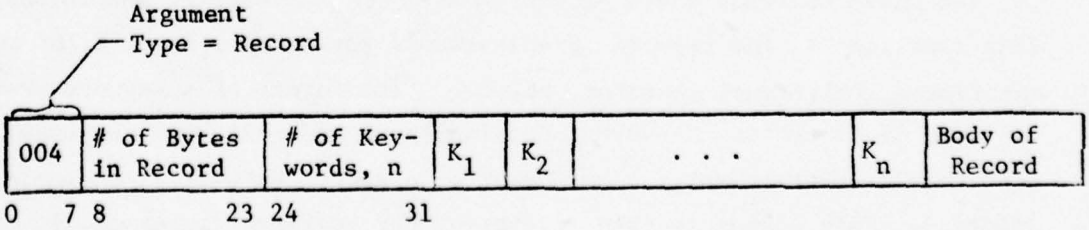


Figure 6. Format of a Record Received by the DBCCP

in Figures 7 and 8, respectively. There are two types of file sanctions recognized by the DBCCP. The first is used in the case when the type A protection is specified for the user. The second type is used in the case when the type B protection is specified for the user. These are shown in Figures 9 and 10. An access descriptor is associated with a file sanction. The format of an access descriptor is shown in Figure 11. All descriptors except the default access descriptor have their file privilege indicators turned off. The file privilege indicators in a default access descriptor (associated with a database capability) are used to indicate the user's access privileges to the entire file. Security descriptors, which are sent by the PES prior to the creation of a file have the format shown in Figure 12. A security descriptor plays a pivotal role in the determination of security atoms. We shall have more to say about security descriptors when we describe the algorithms in the CCRP. Pointers (see Figure 13) are used by the PES to retrieve information from a particular area of the MM.

2.3.2 Command Formats

There are seventeen basic commands which are recognized by the DBCCP. This command set is by no means exhaustive, but is, nevertheless, complete in the sense that it enables the PES to take advantage of all of the facilities that are provided by the DBC. In any particular implementation, this command set may be augmented for ease of use or to correspond to additional facilities incorporated in the DBC. All the commands have the general format <command ID, command code, priority, user ID, argument set>. The command ID is used to uniquely identify a command as it is processed by various components of the DBC. Since we do not anticipate that the DBC will ever process more than 256 commands at a time, this field is chosen to be 8 bits. The command code is used to indicate the service needed. The priority field indicates the level of service requested. Seven levels of priority (1-7) may be specified. The higher the priority number the better the service time is likely to be. The priority numbers may be used to distinguish batch jobs from interactive requests. The user ID and file ID identify the user requesting the service and the file upon which the request is to be carried out. The argument set carries arguments which are needed to identify the data within the file or to provide information about the file.

The open-database-file-for-creation command is required to be sent to the DBCCP before records of the file are loaded into the DBC. See Figure 14.

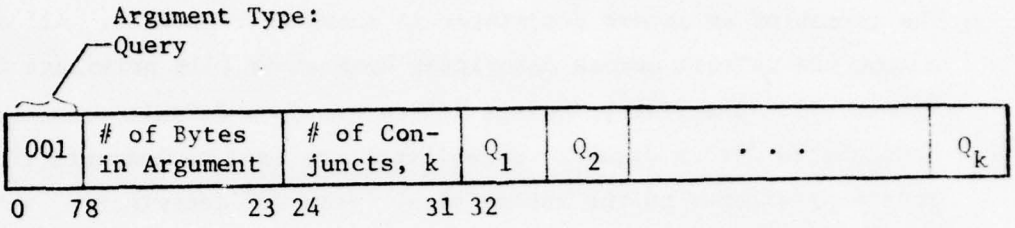


Figure 7. Format of a Query Received as an Argument of a PES Command

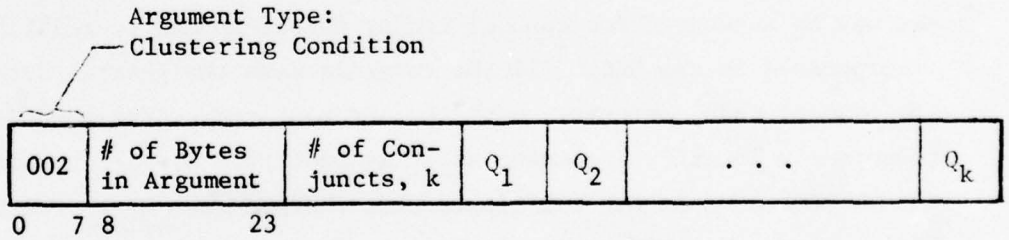


Figure 8. Format of a Clustering Condition Received as an Argument of a PES Command

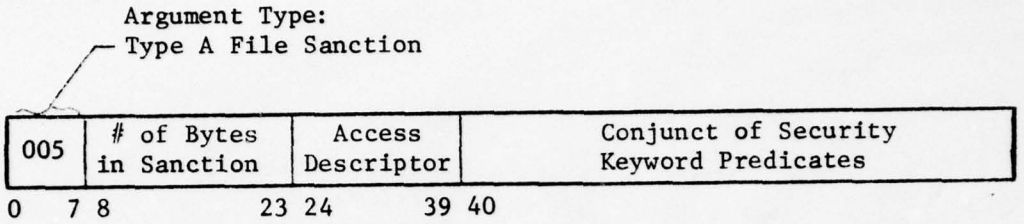


Figure 9. Format of a Type A File Sanction Consisting of a Conjunct of Security Keyword Predicates and an Access Privelege of 16 Bits

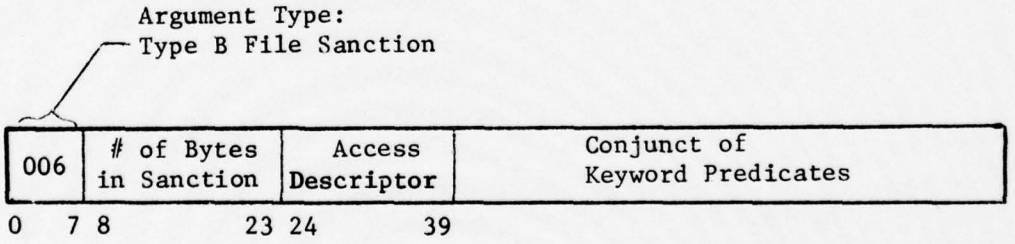
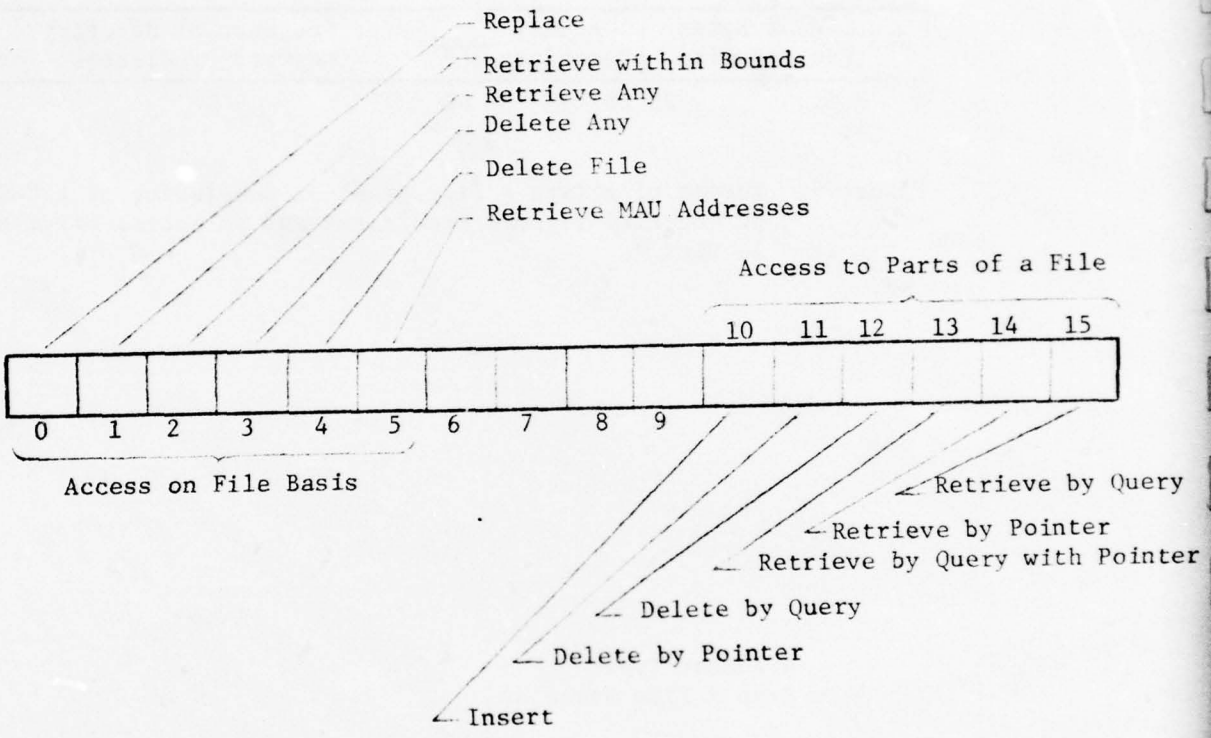


Figure 10. Format of a Type B File Sanction Consisting of a Conjunct of Keyword Predicates and an Access Privilege Set of 16 Bits



Note: A '1' in a bit position indicates a right to perform the access, while a '0' indicates a denial of the right.

Figure 11. Format of Access Descriptor

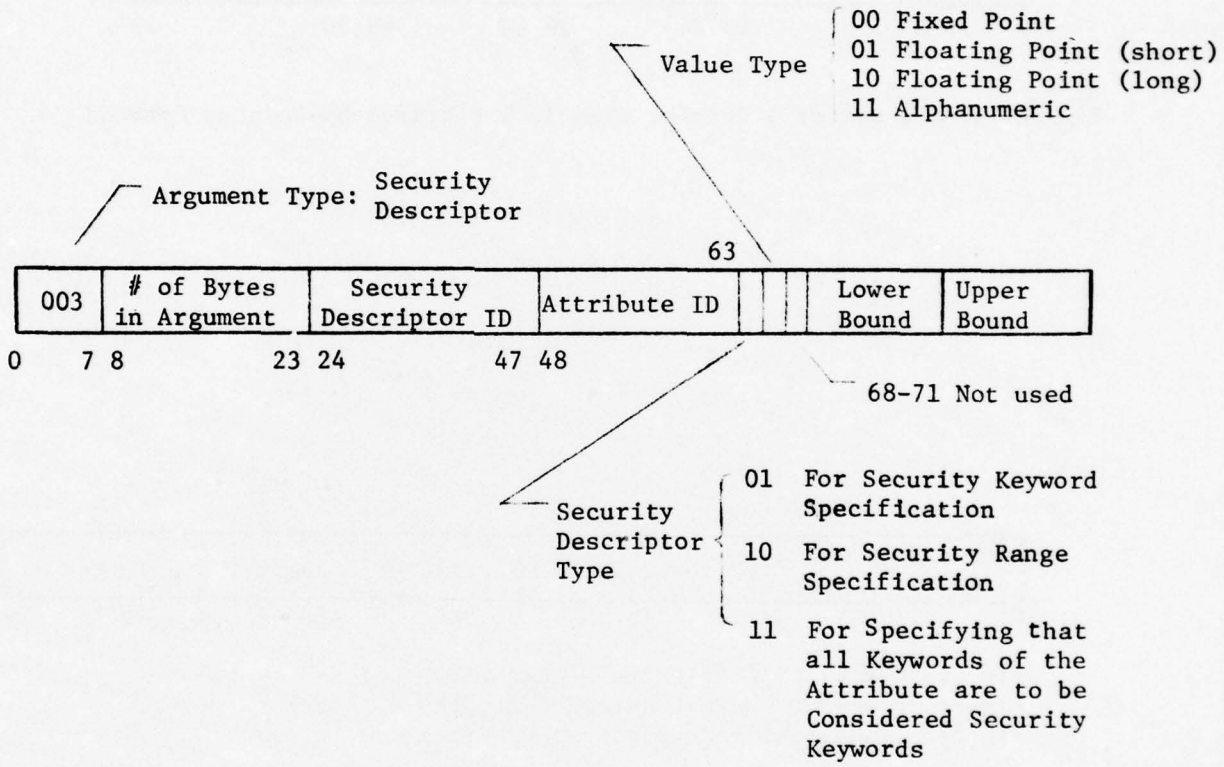


Figure 12. Format of a Security Descriptor Received as an Argument of a PES Command

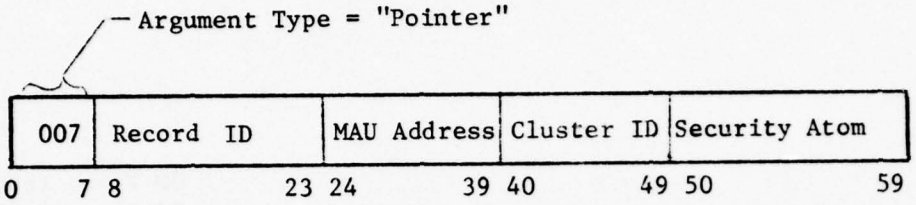
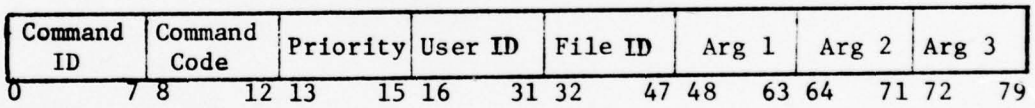


Figure 13. Format of a Pointer Used in a Retrieve-by-Pointer Command



Arg 1: Number of Attributes Needed (16 Bits)
Arg 2: Number of MAUs Required Initially (8 Bits)
Arg 3: Additional MAUs Required (8 Bits)

Command Code = 01

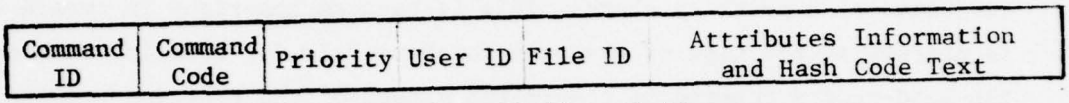
Priority: 1-7 (Higher priority numbers command faster service)

Figure 14. Format of Open-Database-File-for-Creation Command

This command provides information on the number of attributes the file is to have, the number of MAUs that need to be allocated initially, and the number of MAUs that may be allocated if the initial allocation is insufficient. Two other commands are needed to provide information on attributes and security descriptors. See Figures 15 and 16. Once these commands are given, the DBCCP is ready to accept records to be loaded into the DBC. This is done by means of the load-record command. See Figure 17. It should be noted that records loaded by this command are not subjected to a security check. This is because the right to create a file is checked at the time of the open-database-file-for-creation command, and the load-record command is considered a part of the creation process. The close-database-file command (See Figure 18) is used to indicate that the file may be deactivated, i.e., to indicate that there will be no more commands from the user on the file.

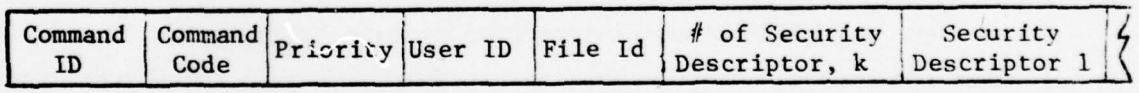
Since the processing for creation of a database file is different from that for accessing a file, a separate command called the open-database-file-for-access (See Figure 19) is provided. This command assumes that a file whose ID is an argument of the command has already been created and is known to the DBCCP. There are certain restrictions when a file is opened either for creation or for access. During creation, a user may not issue any access commands. The only commands permissible are the 3 load commands shown in Figures 15, 16, and 17. During file access, a user may not issue a load-record or load-security-descriptor command. He is, however, required to issue a load-attribute-information command following an open-database-file-for-access command. This is necessary, because the keyword transformation unit (KXU) discards attributes information when a file is closed [2]. The access commands that may be issued when a file has been opened for access will now be described.

There are four commands that may be used to retrieve records of a file. The retrieve-by-query command (see Figure 20) will probably be the most common type of retrieval request. In this command, a query made of keyword predicates in the disjunctive normal form is used to identify records desired by the user. The retrieve-by-pointer command (see Figure 21) is used by the sophisticated user who knows exactly where the desired record is stored. The use of this command generally implies a greater privilege accorded to the user than those who can merely use the retrieve-by-query command. Furthermore, the processing of this command is less involved than



Command Code = 02_8

Figure 15. Format of a Load-Attribute-Information Command



Command Code = 03_8

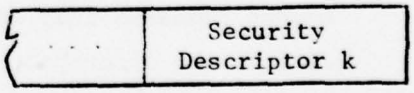


Figure 16. Format of the Load-Security-Descriptor Command

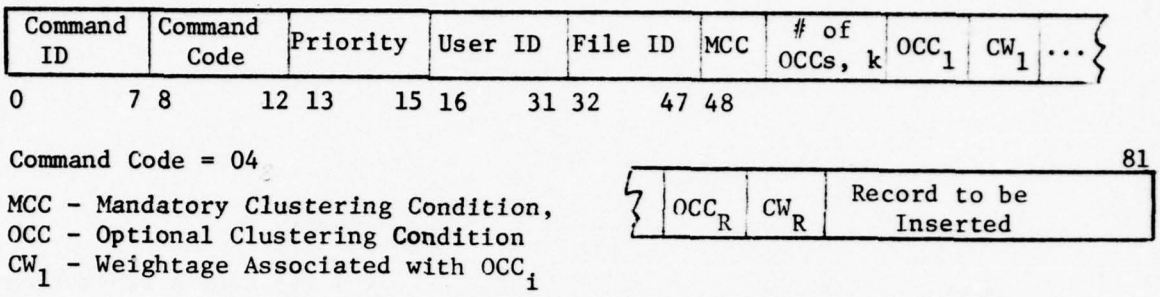
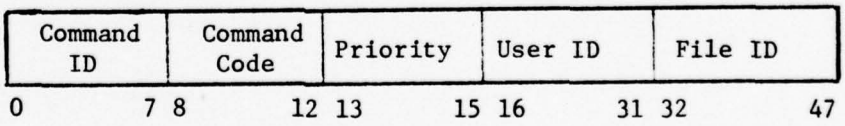


Figure 17. Format of the Load-Record Command



Command Code = 05_8

Figure 18. Format of the Close-Database-File Command

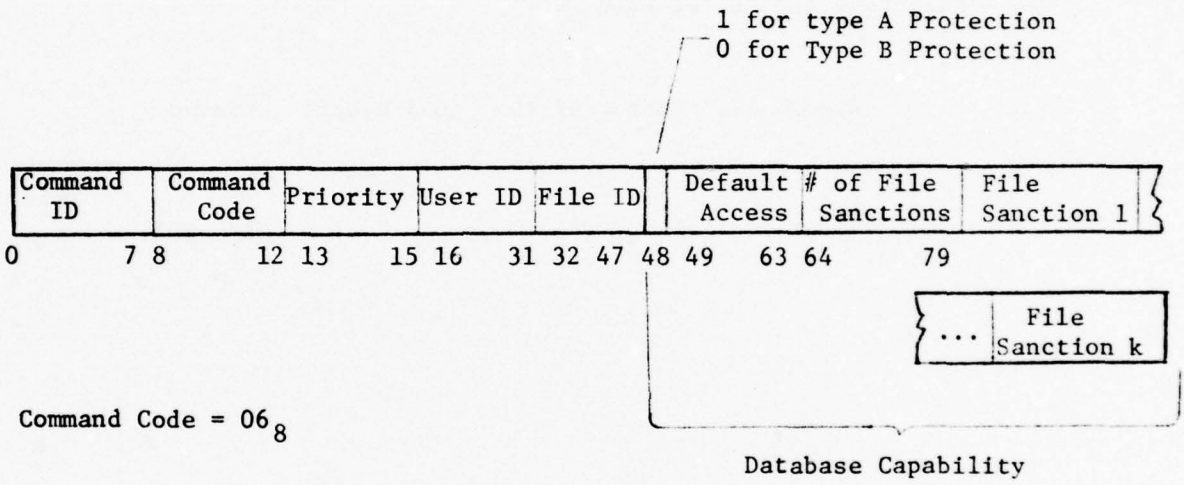
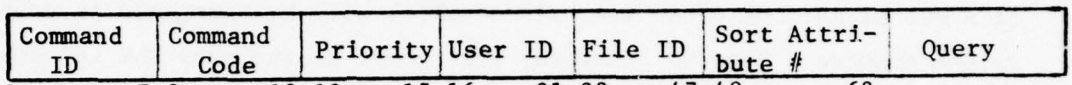
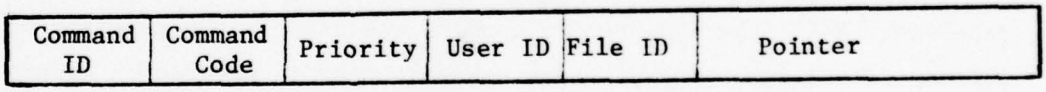


Figure 19. Format of a "Open Database File for Access" Command



Command Code = 07_8

Figure 20. Format of a Retrieve-by-Query Command



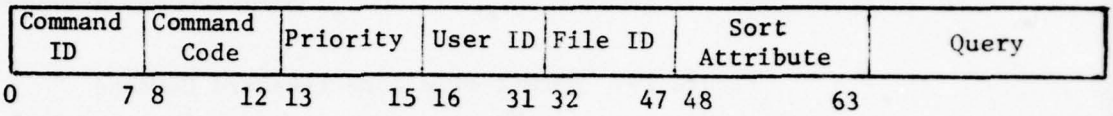
Command Code = 10_8

Figure 21. Format of a Retrieve-by-Pointer Command

the retrieve-by-query command. This is because directory information need not be obtained from the structure memory (SM) to determine the MAU address(es). The retrieve-by-query-with-pointer command (see Figure 22) is provided so that the user may determine the pointer values of a set of records satisfying a query and use these pointers as arguments in subsequent retrieve-by-pointer commands. The fourth retrieval command is called the retrieve-within-bounds command (see Figure 23). This command requires two pointers as arguments. The two pointers are used as lower and upper bounds of a set of records of a file, all of which will be retrieved in response to the command. The two pointers, must point to the same MAU address. They may differ only in the record numbers. Cluster identifiers and security atom names in the pointers are ignored. Record numbers are unique only within an MAU, therefore, it would be meaningless to specify pointers into different MAUs. The use of this command implies a level of access privilege that is higher than that of the other three commands. The right to use this command is indicated in the default access descriptor of the database capability (see Figures 19 and 11). This command is particularly useful when the user wants to process all the records in a file (or in an MAU) but has workspace only for a small fraction of the total number of records. All the retrieval commands except the retrieve-by-pointer command have a sort option, i.e., the records that are retrieved may be sorted on the values of one attribute.

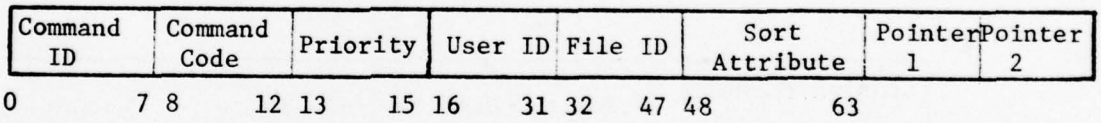
The insert-record command (see Figure 24) is used to add records to a database. While the load-record command was used in the creation process for rapid loading of the database, this command is used as an update command. Thus, this command undergoes the same kind of security check as do other access commands. The load-record command may not be used in place of an insert-record command.

There are three deletion commands recognized by the DBCCP. The delete-by-query command (see Figure 25) is very similar (in processing) to the retrieve-by-query command. It uses a query to identify those records that have to be removed from the file. The delete-by-pointer command (see Figure 26) is similar to the retrieve-by-pointer command and is used to delete a sepcific record. Quite often a user may wish to destroy the entire file. This action is provided by the delete-file command (see Figure 27). This command not only releases the database areas (MAUs) occupied by the file, but also the SM space occupied by keyword directory entries and auxiliary



Command Code = 11_8

Figure 22. Format of a Retrieve-by-Query-with- Pointer Command



Command Code = 12_8

Figure 23. Format of a Retrieve-Within -Bounds Command

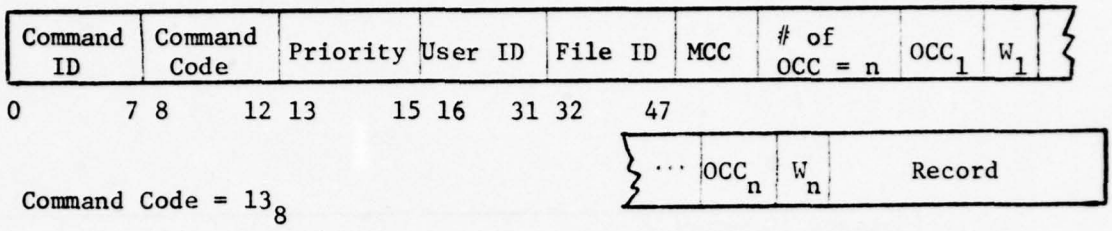


Figure 24. Format of a Insert-Record Command

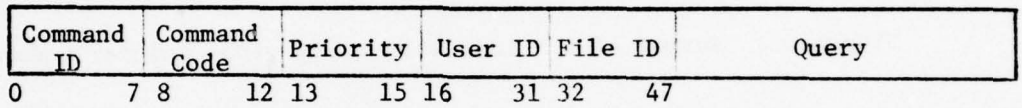


Figure 25. Format of a Delete-by-Query Command

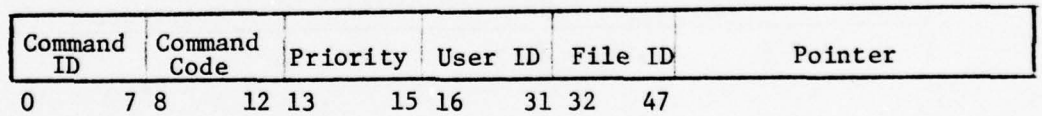


Figure 26. Format of a Delete-by-Pointer Command

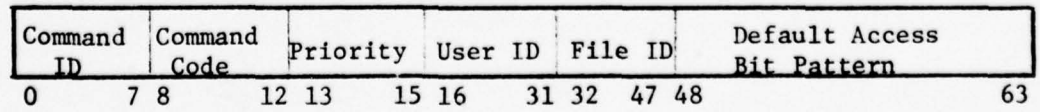


Figure 27. Format of a Delete-File Command

information kept by the DBCCP. It should be noted that the delete-file command does not involve file access, and, therefore, it does not require an open-database-file-for-access command to precede it. Furthermore, at the time this command is given, the file should not be open for access by some other user. If it is, then the command will be rejected by the DBCCP.

In database operations, it is frequently desired to update certain fields of a record and retain only the updated version of the record. Such a facility is provided for by the replace-record command (see Figure 28). There are two arguments to this command - a pointer to the old record that is to be replaced and the new record that is to replace the old record. Internally, this command is divided into two parts - a delete-by-pointer for the old record, and an insert-record command for the new record. A user who wishes to use this command must have the privilege of replacing records in the file as indicated in the default access descriptor (see Figure 11). Furthermore, he should have the right to delete (by pointer) the record he is replacing and the right to insert the new record. Thus a replace-record command requires the user to have three kinds of access rights - record replacement on a file basis, deletion by pointer and insertion of record. This interpretation of the replace-record command is motivated by two factors: first, we should enable the user to change any part of the record he wants to replace; this includes values of security keywords as well. Second, given the first factor, it would be illogical to associate the replacement privilege either with the security atom containing the old record or with the security atom containing the new record.

Earlier, we described the retrieve-within-bounds command which was useful in bulk processing. However, in order to use this command, a user needs to know the MAU addresses in which his file is located. This knowledge can be obtained by using the retrieve-MAU-addresses command, whose format is given in Figure 29. The DBC responds by providing the user with a list of MAU addresses in which the file, identified in the command, is located. Finally, the load-creation-capability-list command (see Figure 30) is used to indicate to the DBCCP the identity of the users who may issue the command open-database-file-for-creation. Table I gives the list of DBC commands.

Command ID	Command Code	Priority	User ID	File ID	Pointer	Record
0 7	8 12	13 15	16 31	32 47		

Command Code = 17_8

Figure 28. Format of a "Replace-Record" Command

Command ID	Command Code	Priority	User ID	File ID

Command Code = 20_8

Figure 29. Format of a Retrieve-MAU-Addresses Command

Command ID	Command Code	Priority	System ID	# of IDs , k	User ID_1	· · ·	User ID_k
0 7	8 12	13 15					

Command Code = 21_8

Figure 30. Format of a Load-Creation-Capability-List" Command

Table I. List of Commands Recognized by DBC

Code	Command Function	Command Type (A - access type P - preparatory type
01	Open Database File for Creation	P
02	Load Attribute Information	P
03	Load Security Descriptor	P
04	Load Records	A
05	Close Database File	P
06	Open Database File for Access	P
07	Retrieve by Query	A
10	Retrieve by Pointer	A
11	Retrieve by Query with Pointer	A
12	Retrieve Within Bounds	A
13	Insert Record	A
14	Delete by Query	A
15	Delete by Pointer	A
16	Delete File	P
17	Replace Record	A
20	Retrieve MAU Addresses	P
21	Load Creation Capability List	P

2.3.3 Table Structures in the DBCCP

A number of information tables are maintained by the DBCCP in order to carry out the various commands issued by the PES. Here we shall discuss several of the tables that are accessed by the CCRP and CTP via the communication bus and the argument-and-structure-loop-response bus (see Figure 3). Five separate memory elements contain these tables. More specifically, the command argument table memory (CATM) contains the argument table. The security information table memory (SITM) carries security descriptors, file sanctions and atomic privilege lists of active users, while the file information table memory (FITM) contains the file information table, the attribute bit map, and the MAU allocation table. The command status table memory (CSTM) holds the status information of commands currently under processing. The database response memory (DRM) holds response data to be sent to the PES. In order for a processor (CCRP, MM, SFP, SLIP or CTP) to access any of these tables, it must first obtain control of the data bus (communication bus or argument-and-structure-loop-response bus) to which the memory unit containing the table is connected. As long as a processor has control of a bus, no other processor can gain access to any of the table memories on that bus. By providing for high bandwidth busses, one can reduce the amount of time for which a processor needs to "lock" up a bus.

The above arrangement for serialization of access to table memories eliminates most of the deadlock situations that might otherwise be anticipated. However, there is one situation that requires consideration. Supposing, two of the processors each have control over one of the two busses and then demand the use of the other bus. In this case the two processors will be waiting indefinitely on one another causing a deadlock. This situation is overcome by making it mandatory on each of the processors to release the bus it is holding before requesting the use of the other bus.

A. The Command Argument Table - This table contains queries, MCCs, OCCs and records that are received as arguments of DBCCP commands awaiting execution. Each command argument occupies a contiguous block of memory. Since entries in the table are of variable size, the following scheme is used to allocate and free memory. A doubly-linked AVAIL list of available memory space is maintained. When an argument is to be placed in the table, the first-fit method [6] is used to allocate sufficient space for the argument. When an argument is to be deleted (because the command to which

the argument belongs has been processed), space occupied by the argument is linked into the AVAIL list after ensuring that (possible) adjacent blocks of free memory are properly collapsed. The AVAIL list is maintained by using a small part of the free spaces themselves. The overhead due to such links is small compared to the size of the blocks that are allocated and freed. As the number of commands in the DBC increases, the argument table will progressively become more and more full, until at some point, the first-fit method will fail to yield enough space for an argument. At this point the DBCCP will stop accepting further commands until more space is freed. Although compaction could be resorted to consolidate all the available pockets of memory space, we have chosen not to, for a number of reasons. First, even if we perform compaction, the acceptance of more commands will very soon exhaust the consolidated available space. Second compaction involves resolution of argument pointers in the command status table. Since each command may have several arguments, the overhead due to resolution of pointers can be significant. Third, even if we don't perform compaction, space will become available when the processing of existing commands in the DBC is completed, thus allowing new commands to be accepted. In Figures 31 and 32 the formats of a free block and an occupied block of memory are shown.

B. File Information Table (FIT). There is one file information table (FIT) for each file known to the system. The FIT is created at the time the file is created. It contains information about space allocated to the file, and certain security related information. The FIT is a variable length table and is composed of a set of fixed size blocks in the table memory. The format of a file information table is shown in Figure 33. An FIT has five fields. The first field contains the file name. The file name is a 16 bit pattern generated by the PES. The second field contains the identity of the file creator and information about its status (i.e., whether the file is active or not, etc.). The third field is a list of attribute identifiers allocated to the file. It is useful to keep such a list; since at the time the file is deleted, we need to reclaim the attribute identifiers used by the file. Field 4 is a list of security descriptor identifiers. These identifiers identify the security descriptors of the file which are stored in another table memory called the security information table memory (SITM). The format of a security

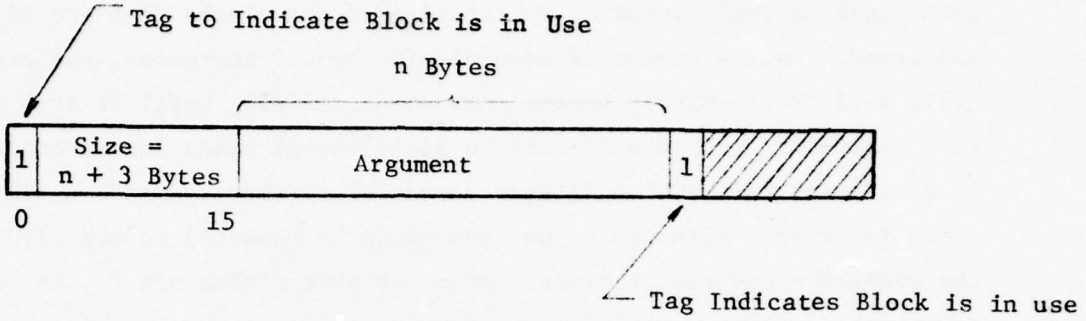


Figure 31. Details of a Block in Use in the Command Argument Table Memory (CATM)

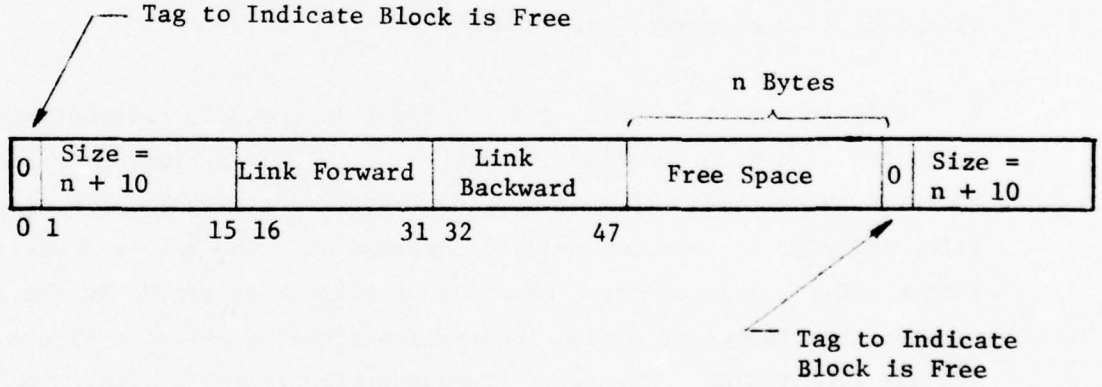


Figure 32. Details of a Free Block in the Command Argument Table Memory (CATM)

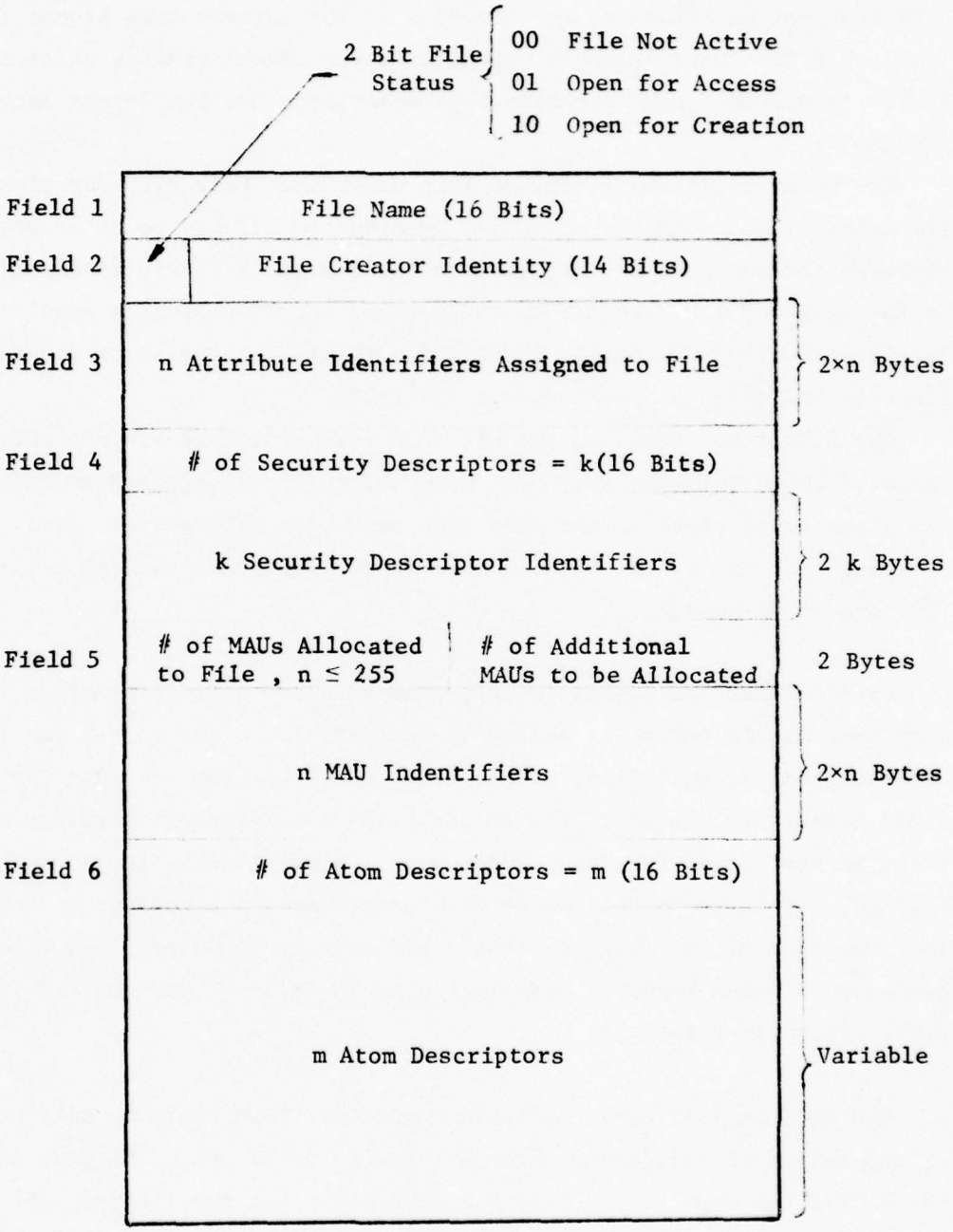


Figure 33. File Information Table (FIT)

descriptor was shown earlier in Figure 12. Field 5 identifies all those MAUs that have been allocated to this file. It also keeps track of the number of MAUs that may be allocated to this file if the current MAUs become full. In field 6 the descriptions of security atoms which actually exist in the field are stored. Each security atom descriptor has the format shown in Figure 34.

The DBCCP maintains a list of file names and their FIT addresses in the FIT memory. This list is consulted whenever an FIT has to be accessed. Although the total number of files in the system is likely to be large, say, in the range of 1,000 to 2,000, the size of the list remains small (4-8K bytes). The file creation capability list, which identifies the users who can create files, is stored in a fixed area in the FITM.

The FITM also contains, two bit maps - an attribute bit map which keeps track of the attributes that have been allocated so far, and an MAU bit map which keeps track of the MAUs that have been allocated so far. The attribute bit map is 64K bits in size (= 8K bytes) and the MAU bit map is 32K bits (= 4K bytes).

C. User Information Tables (UITs). There is one user information table for each user who is currently active in the DBC, i.e., for a user who is accessing one or more files in the DBC. A UIT has two fields. The first field identifies the user, the second field has information regarding the files he has opened for creation/access. In particular, the field has the file ID, a pointer to the set of file sanctions and a pointer to the privilege list for each of the files currently accessed by the user. The file sanctions are stored in the security information table (SIT). The size and format of the UIT is shown in Figure 35.

D. The MAU Space Information Table (MAUSIT). This table is used to keep track of the amount of available space in the MAUs of the MM. There is one entry in the MAUSIT for each MAU in the MM. Each entry has two fields. The first field contains the number 8-byte blocks available in the MAU. This field occupies 2 bytes and can, therefore, represent 2^{16}-1 blocks (≥500K bytes). The second field carries information about the space available in the track with the largest capacity. The space is measured in terms of the number of complete sectors (1 sector = 128 bytes). The field occupies one byte, thus providing for a

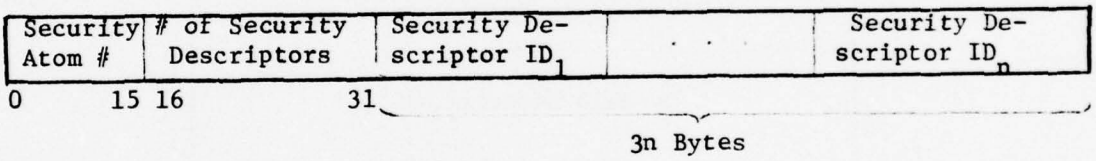


Figure 34. Format of a Security Atom Descriptor in FIT

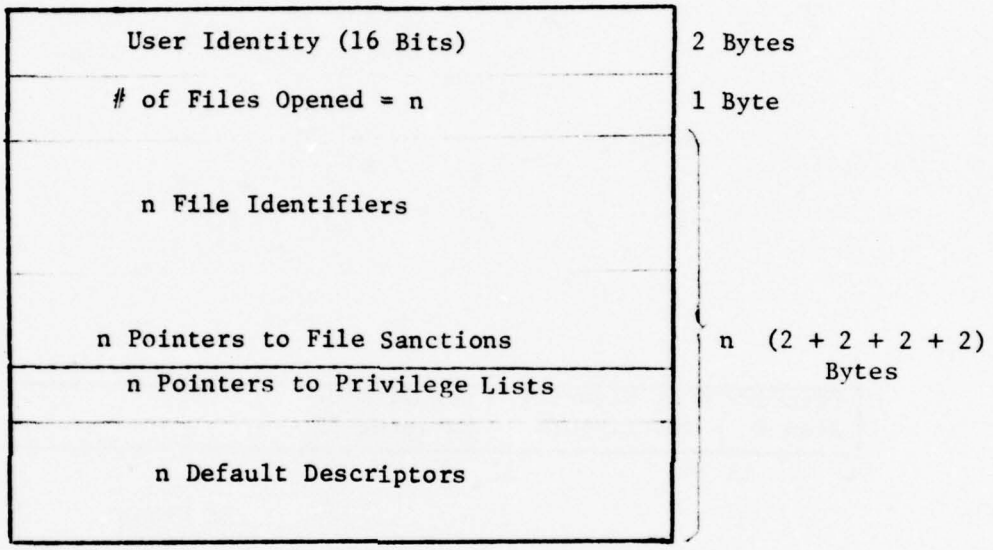


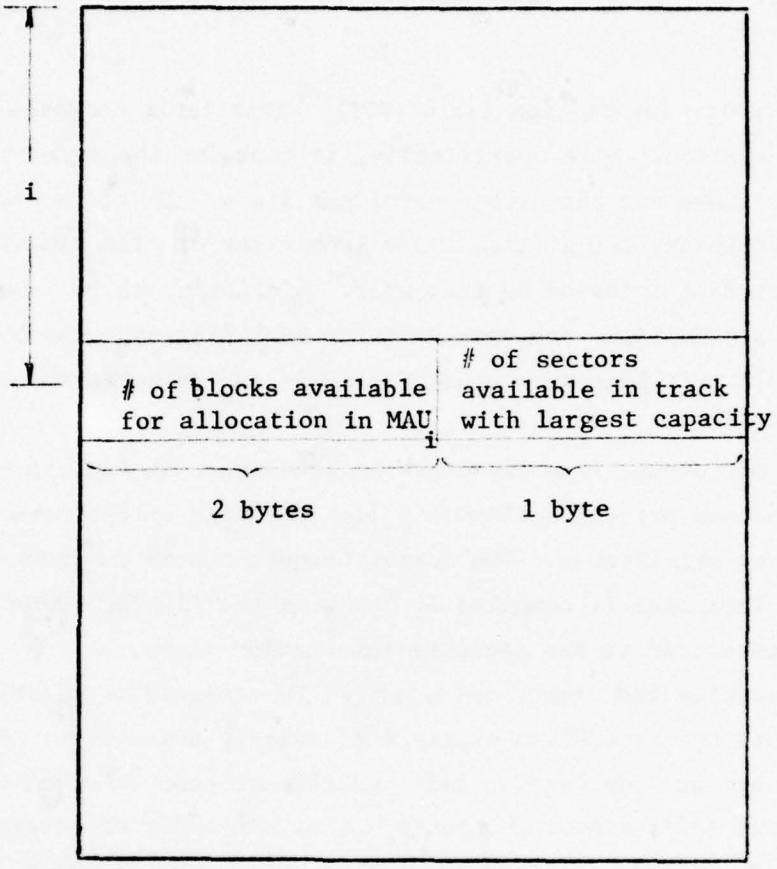
Figure 35. Format of a User Information Table (UIT)

maximum track capacity of 32K bytes. Each entry is, therefore, three bytes long. For a database of 10^9 - 10^{10} bytes, we need between 2,500 to 25,000 MAUs. Thus, the maximum number of entries is 25,000. The required table memory is about 75K bytes. The format of a table entry is given in Figure 36.

E. The Security Information Table (SIT). This table contains all security related information. More specifically, it contains the security descriptors, the file sanctions and the atomic privilege lists. There is one set of security descriptors for a file, there is one set of file sanctions for each user for each file accessed by that user. Similarly, there is one list of atomic access privileges for each user for each file accessed by the user. (Of course, atomic privileges are created only if the user has the type A protection.)

The format of the file sanction has been shown in Figures 9, 10 and 11. The atomic access privilege list is a list of pairs of the form <security atom number, access privileges>. The format of such a pair is given in Figure 37. This list is compiled at the time the file is opened for access and is stored in the security information table.

Each security descriptor has a unique ID assigned by the PES. This number is used by the CCRP to access the security descriptor. Security descriptors are used during the file creation process in order to identify security atoms defined by the input records. In order to access security descriptors rapidly during this process, a small part of the SITM is used as an access vector. The n low order bits of a security descriptor are used to index the access vector. An entry in the access vector heads a list of security descriptors all of which have the same n low order bits. The value of n depends on the level of performance required of the scheme. Typically, the value of 2^n could be in the range of one-sixteenth to one fourth the anticipated number of security descriptors ($\leq 2^{16}$) in the system. The remaining part of the table memory is dynamically allocated to contain security descriptors, file sanctions and atomic privilege lists. File sanctions and atomic privilege lists do not require access vectors, since the users' UITs point to the respective file sanctions and atomic access privilege list. In Figure 38 the organization of the security information table is illustrated. Blocks of free and allocated memory are kept track of in a



Note: One entry per each MAU
1 block on an MAU = 8 bytes; 1 sector on an MAU = 16 blocks

Figure 36. Format of an entry in the MAU Space Information Table (MAUIT)

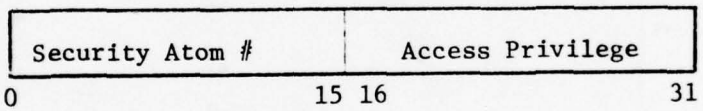
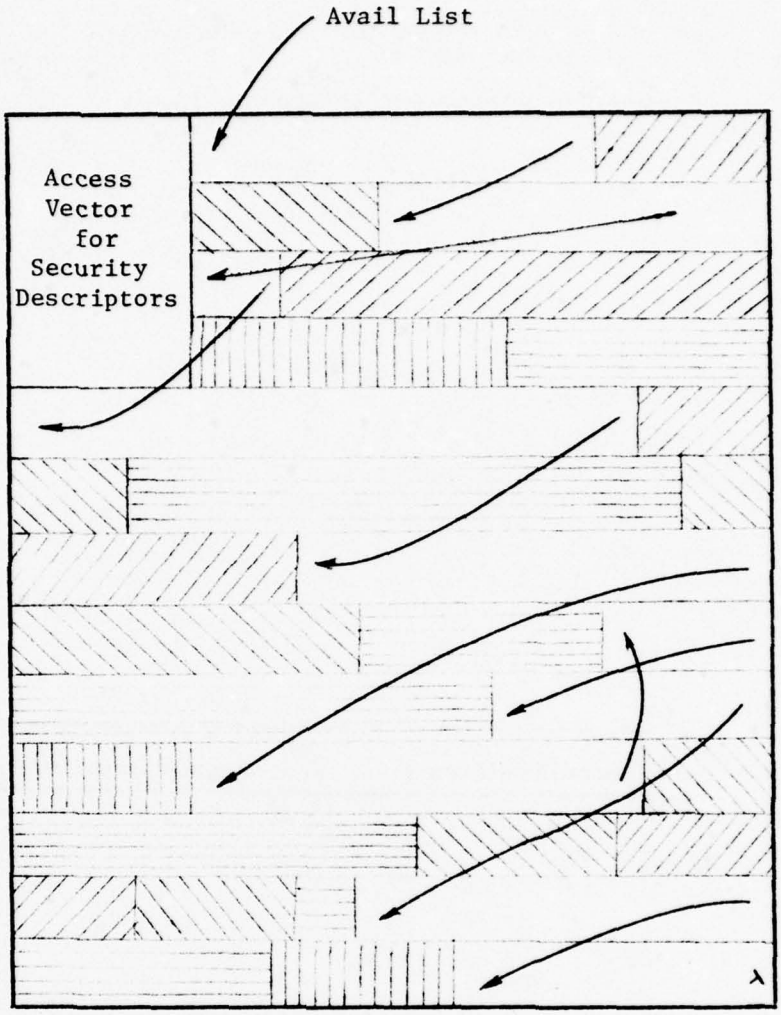


Figure 37. Format of an entry in The Atom Access Privilege List



Security Descriptors

Type A File Sanctions

Free Blocks

Atomic Privilege List

Type B File Sanctions

Note: Each entry in the access vector is the head of a list of security descriptors. All descriptors in a list have the same high order 8 bits in their identifier value.

Figure 38. Security Information Table Memory

manner similar to the scheme used in the command argument table (i.e. maintenance of AVAIL list etc.). However, unlike blocks of allocated (free) memory in the command argument table, blocks of allocated (free) memory in the security information table are stabler, in the sense that they are not released (allocated) as frequently as in the command argument table.

F. The Command Status Table (CST). This table is maintained by the DBCCP to indicate the status of the commands accepted by it. The table memory is divided into three parts (see Figure 39): a table of priority list headers (PLH), a table of status information (SIT), and a table of argument pointers (APT). There is one entry for each priority level in the table of priority list headers. An entry in this table points to an entry in the status information table. Each command under processing has an entry in the SIT (see Figure 40). All entries of commands at the same level of priority are chained together with the appropriate entry in the PLH pointing to the first entry in the chain. An entry in SIT has ten fields:

Field 1: An 8 bit status field reflects the progress of the command through the DBCCP.

Field 2: An 8 bit field identifies the command to the DBC.

Field 3: An 8 bit field specifies the command function.

Field 4: A 16 bit field identifies the file referred to by the command.

Field 5: A 16 bit field identifies the user who caused the PES to issue this command.

Field 6: This 16 bit field stores the address of the control memory of the processor (CTP or CCRP) waiting for SLIP to provide structure information.

Field 7: This 8 bit field records the number of MM orders issued by the CTP.

Field 8: This 8 bit field records the number of security violations that are encountered in the execution of the command.

Field 9: This field points to the APT where pointers to the arguments table are stored. There is one pointer in the APT for each command argument.

Field 10: This field points to the entry of a command at the same priority level as this command.

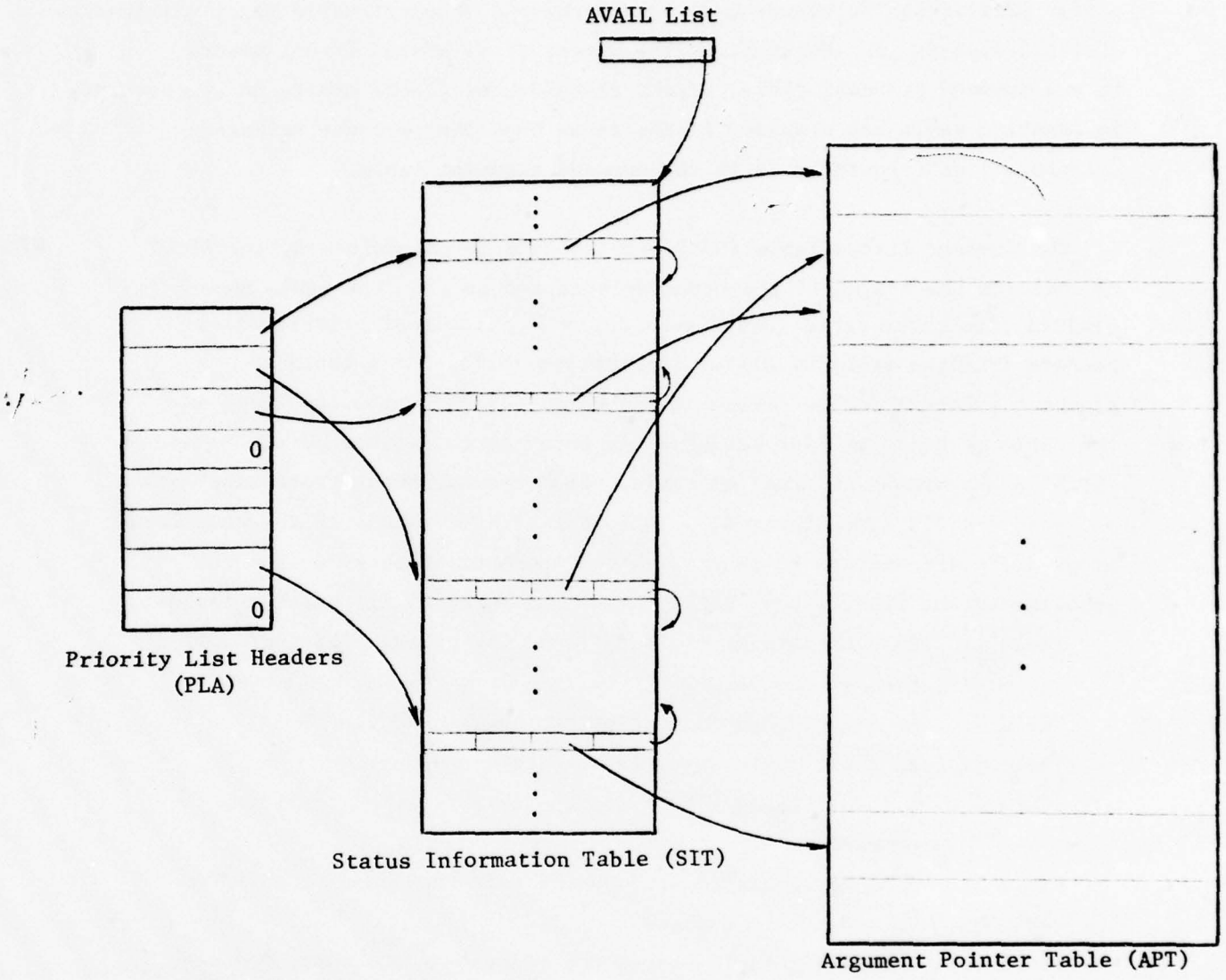


Figure 39. Organization of the Command Status Table (CST)

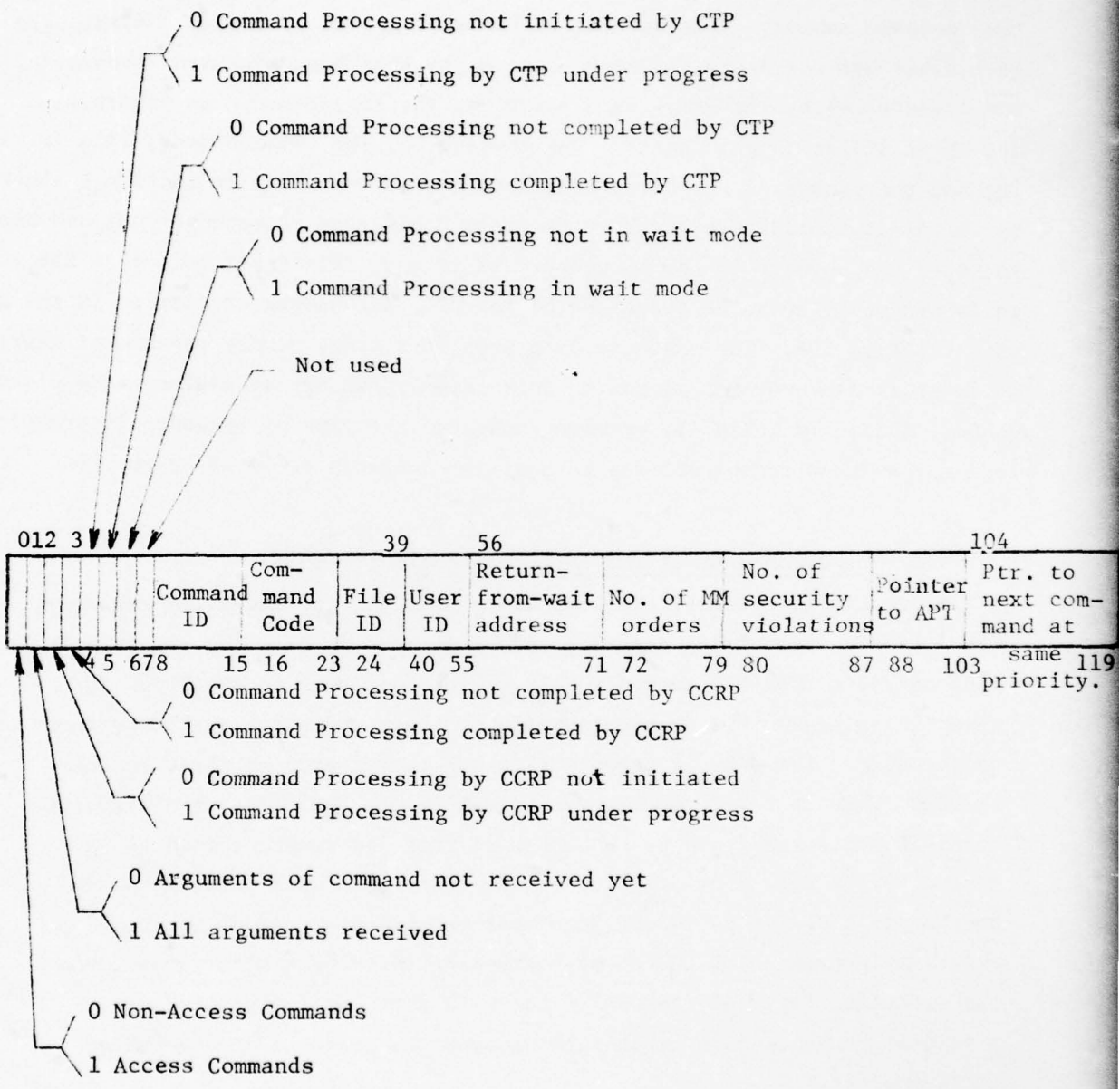


Figure 40. Format of an entry in the SIT

G. The Database Response Memory (DRM) - This memory is used primarily by the SFP to store records retrieved by the MM. It is also used by the DBCCP to store accept/reject signals for non-access commands. The memory consists of two parts - the response summary table and the response data area (see Figure 41a). The summary table has one entry for each command that has been executed (successfully or unsuccessfully) by the DBC. Each entry has the format shown in Figure 41b. There are seven fields in each entry. The command ID, the command code, file ID, user ID, and the number of security violations are as described in Section F above. There is one field to indicate whether the command has been accepted or not and one field to store the pointer to the response data if any. Since not more than 256 commands are expected to be processed by the DBC, the number of entries in the summary table is 256. The response data area is a large memory capable of storing at least as many records as can be processed by the SFP at a time (see discussion in Sec. 4.3). In Table II, we have indicated the type of response information that may be sent to the PES for each of the commands accepted by the DBC.

2.4 The Command Check and Response Processor (CCRP)

As mentioned earlier, the CCRP is responsible for receiving commands from the PES, performing security checks on them if possible and routing response data from the security filter processor back to the PES. The algorithm given in this section pertain to these and other auxiliary functions of the CCRP. The CCRP is capable of being interrupted by three sources - the PES when it has a command for execution, by the SLIP when structure information is ready and by the SFP when there is response data to be routed to the PES. The priority of the interrupts with respect to one another is a design decision which must be made by considering the relative importance of the three sources of interrupts with respect to the execution logic of the CCRP. Later in this section we propose a priority scheme which takes into account the relative importance of the three sources.

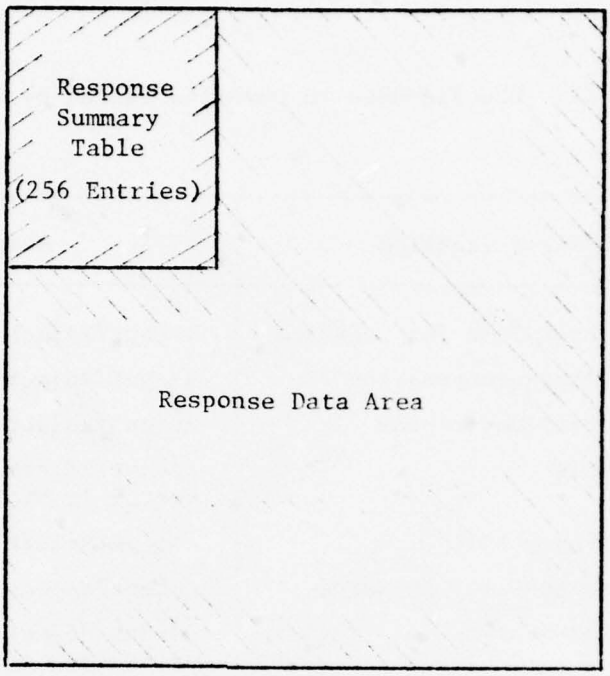


Figure 41a. Database Response Memory

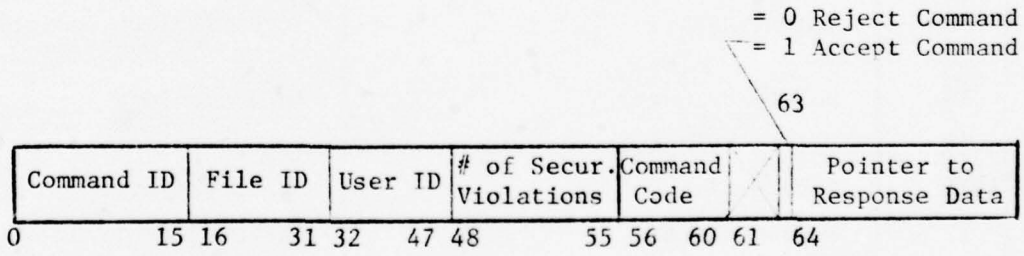


Figure 41b. Format of an Entry in the Response Summary Table

Table II. DBC Response to Commands Issued by PES

Code	Command Function	Response
01	Open Database File for Creation	Accept/Reject
02	Load Attribute Information	Accept/Reject
03	Load Security Descriptor	Accept/Reject
04	Load Records	Accept/Reject, pointer to record in MM
05	Close Database File	Accept/Reject
06	Open Database File for Access	Accept/Reject
07	Retrieve by Query	Accept/Reject, records without pointers
10	Retrieve by Pointer	Accept/Reject, record without pointer
11	Retrieve by Query	Accept/Reject, records with pointers
12	Retrieve within Bounds	Accept/Reject, records without pointers
13	Insert Record	Accept/Reject, pointer to record in MM
14	Delete by Query	Accept/Reject
15	Delete by Pointer	Accept/Reject
16	Delete File	Accept/Reject
17	Replace Record	Accept/Reject, pointer to new record in MM
20	Retrieve MAU Addresses	Accept/Reject, addresses of MAUs
21	Load Creation Capability List	Accept/Reject

2.4.1 Security Related Processing

In this section we present algorithms used to create atoms, and algorithms for performing type A security checks.

ALGORITHM A - To extract a security atom from a record for insertion.

Input Arguments: 1. The record to be inserted with k security keywords
2. The security descriptor identifiers from FIT

Step 1: Retrieve all security descriptors of the file from the SIT. Call the set of security descriptor SD. The j-th member of SD is denoted by SD_j.

Step 2: Let the attribute identifier of the i-th security keyword be called A_i and the value of the i-th security keyword be V_i. Set i←1, p←1.

Step 3: Compare A_i with the attributes of the members of the set SD. Call the set of security descriptor with matching attributes SD'. Let there be n elements in SD'.

Step 4: j←1; [In step 5 we examine the j-th security descriptor in SD'].

Step 5: If security descriptor type specifies that the descriptor is a keyword (see Figure 12) then compare V_i with value of the descriptor. If security descriptor type specifies that descriptor is a range of values, then compare V_i with the lower bound and upper bound of range. If the comparison is successful, then go to step 7. If comparison is not successful, then go to step 6. If security descriptor type specifies that all values of the attribute are to be treated as a security keyword then go to step 8.

Step 6: j←j+1; If $j \leq n$, go to step 5; else, i←i+1. If $i \leq k$, go to step 3; else, go to step 10.

Step 7: [Security Descriptor describing keyword is found]. ATOM[p]←SD'[j].ID [In this step, we merely record the security descriptor identifier of the p-th element of the atom describing the record. (see Figure 34)]. Go to step 9.

Step 8: [Each value of the attribute is a security descriptor]. Compare the values of other security descriptors with the value V_i of security keyword from the record. If a match occurs, then set j to the identifier of matching descriptor in SD'; go to step 7. If match does not occur, then

$$\text{ATOM}[p] \leftarrow \text{SD}'[j].\text{ID} + \left\{ \begin{array}{l} \text{low order 8 bit of} \\ \text{the ID of the last} \\ \text{descriptor in SD' with} \\ \text{the same high order 16} \\ \text{bits as SD'[j].ID} \end{array} \right\} + 1$$

Add to SD the security descriptor defined by ATOM[p]. (see notes below)

Step 9: p←p+1; i←i+1. If $i \leq k$, go to step 3; else, go to step 10

Step 10: [ATOM[1] through ATOM[p-1] contains the description of the atom to which the record belongs. If p=1, then it signifies that no security descriptor describes the record, and therefore the record belongs to security Atom 0]. Terminate.

Response - Security atom description in ATOM[1] to ATOM[p-1].

NOTES: Step 8 implements an important facility provided by the DBC. A security descriptor, as explained earlier, may specify that all keywords of a particular attribute must be considered as security descriptors. This specification enables a creator of a file to convey to the DBC that keywords of an attribute are security sensitive without having to actually know the values of the keywords.

ALGORITHM B - To determine the access privileges accorded to a security atom by a set of type A file sanctions.

Input Arguments:
1. A set F of n file sanctions each of which is in the format shown in Figure 9
2. A security atom descriptor as shown in Figure 34.
3. A set of security descriptor SD

Step 1: From the set SD of security descriptors, extract the set SD' of security descriptors defined by the argument security atom.

Step 2: Form an access descriptor A. Set A←ε' [ε' denotes a bit pattern of all 1s]. Set AFLAG←0.

Step 3: Denote the i-th member of F by s_i. Let there be K_i predicates in s_i.

Step 4: [Test if s_i is applicable to argument atom] j←1.

Step 5: Check if T_{ij} is covered by any of the security descriptors in SD'. [By covered, we mean the following: If T_{ij} is an equality predicate, then a security descriptor - which specifies the keyword in T_{ij} - covers T_{ij}. If T_{ij} is a '$\geq$' predicate, then a security descriptor-which specifies either a range of values and whose lower bound is greater than or equal to the keyword value of T_{ij} or a particular value greater than or equal to the keyword value of T_{ij} - covers T_{ij}. If T_{ij} is a '>' predicate, then a security descriptor-which specifies either a range of values and whose lower bound is greater than the keyword value of T_{ij} or a particular value greater than the keyword value of T_{ij} - covers T_{ij}. Similarly, if T_{ij} is '$\leq$' predicates, then a covering security descriptor would be either one which specifies a range of values with the upper bound being less than or equal to the keyword value of T_{ij} or one that specifies a keyword with a value less than or equal to the value of T_{ij}. If T_{ij} is a '<' predicate then a covering descriptor would be defined exactly as for '$\leq$' precidate except the defining conditions would be strictly "less than" instead of "less than or equal to". The '$\neq$' predicate is covered if there does not exist a

security descriptor which either specifies a range of values including the keyword value of the predicate or which specifies a keyword value equal to the keyword value specified in the predicate].

Step 6: If T_{ij} is thus covered, then $j \leftarrow j+1$. If $j > K_i$, then go to step 7. If $j \leq K_i$, then go to step 5. If T_{ij} is not covered, then go to step 8.

Step 7: [file sanction s_i is applicable to atom]. $A \leftarrow A \wedge A_i$ where A_i is the access descriptor associated with s_i. AFLAG$\leftarrow$1. $i \leftarrow i+1$. If $i \leq n$, go to step 4; else, go to step 9.

Step 8: [file sanction s_i is not applicable to atom]. $i \leftarrow i+1$. If $i \leq n$, go to step 4; else, go to step 9.

Step 9: If AFLAG=0, then $A \leftarrow A \wedge A_{def}$, where A_{def} is the default access privilege descriptor provided in the data base capability. Terminate.

Response: 'A' holds the access privilege accorded to the argument atom.

Notes: The crucial step in this algorithm is step 5. Although the definition of "covering" seems formidable, efficient comparison hardware can be constructed to execute step 5.

ALGORITHM C: To determine if access 'a' for a query conjunct Q is allowed by the atomic access privilege list.

Input Arguments:
1. A query conjunct Q in argument table
2. The access type requested 'a'
3. The atomic access privilege list L.
4. The pointer to entry in CST
5. The request identification R-ID

Step 1: Obtain control of the argument and response bus in preparation for issuing a request to the SLIP.

Step 2: Issue the request. Retrieve-security-atom(s)-for-query-conjunct to the SLIP with the arguments Q, priority P (from command status table), request identifier R-ID, file ID (from command status table).

Step 3: Wait for response set from SLIP. [At this point control goes back to the scheduling algorithm].

Step 4: [Control comes here after interrupt from SLIP]. If response set S is empty, then set REJECTFLAG to 1.

Step 5: If response set S is non empty, then, for each member s of S, do step 6.

Step 6: Run down the atomic access privilege list L and extract the access pattern from s. If s is not found in L, then invoke algorithm B with the following arguments: descriptor of atom s from FIT, file sanctions of user for the file and the security descriptors for the file from SIT. Add response of algorithm B to L. If 'a' is not in the access descriptor for s, then delete s from S.

Step 7: Terminate.

Response: Query a is permitted on atoms in set S.

2.4.2 Command Execution

The set of CCRP algorithms is used to process commands from the PES.

An interrupt from the PES signifies that it has a command that needs to be executed by the DBC. There is one algorithm in this set for each of the command that is recognized by the DBC. Control is given to the appropriate algorithm by the scheduling algorithm after the command has been entered in the CST and the arguments stored in the CATM by a first-level interrupt handling algorithm (see Section 2.4.3).

ALGORITHM D: To process the command, open-the-database-file-for-creation.

Input Arguments: 1. The CST entry in which the command is stored.
2. Three arguments shown in Figure 14.

Step 1: Extract user ID from CST entry and check against creation capability list. If user does not have the right to create a file, send reject signal to PES and go to step 5.

Step 2: Find the number of attribute identifiers requested (argument 1 in Figure 14). Determine from the attribute bit map if there are enough attributes identifiers to meet the request. If there are, then send identifier values back to PES along with the command ID. If not, give a reject signal to PES, and go to step 5.

Step 3: Create a file information table in the FITM. Also create a user information table in the FITM, [Since the FIT and UIT are variable length tables, this step allocates one block; later more blocks may be acquired]. Store the attribute identifier values allocated to the file in the FIT.

Step 4: Determine if the number of MAUs requested for initial allocation to the file is available. (Use MAU allocation bit map). If there are enough MAUs, then record the addresses of the allocated MAUs in FIT, set status bits in FIT to indicate that the file is now open for creation. Store the number of MAUs allocated and the additional number of MAUs that can be allocated, in the FIT. If the number of MAUs requested to be allocated initially is not available, send reject signal to PES and go to step 5. Else terminate.

Step 5: Release space occupied by arguments in CATM, remove command from CSTM and terminate.

ALGORITHM E: To process the command, Load-attribute-information.

Input Arguments: 1. CST entry
2. The pointers to arguments in CATM.

Step 1: Check (from FIT) if the file is open for creation. If file is not open for creation, send reject signal to PES and go to step 7.

Step 2: Obtain control of the argument and response bus in preparation for issuing a request to the SLIP.

Step 3: Issue the request, load-hash-algorithms-for-a-file to the SLIP with the following arguments: file-ID, request ID, pointer to hash algorithm in the argument table memory, and priority of request from command status table.

Step 4: For each attribute in the arguments, issue the request, load-attribute-information-for-an-attribute to the SLIP with the

following arguments: file ID, request ID, pointer to the attribute information, the attribute identifier and the priority of request.

Step 5: Wait for response from SLIP [At this point control is given to the scheduling algorithm. The wait status bit is turned on to indicate that the command processing is halted until the SLIP responds; the return address points to step 6].

Step 6: If the KXU has rejected any of the above requests because of lack of space, then send reject signal to PES, release argument table space, FIT, UIT, and MAUs allocated in algorithm D. If the KXU has accepted the requests then send an accept signal to the PES and release argument table space only.

Step 7: Remove command from command status table. Terminate.

ALGORITHM F: To process the command, load-security-descriptor.

Input Arguments: 1. CST entry
2. The pointer to CATM where the security descriptor are stored.

Step 1: Check (from FIT) if file is open for creation. If file is not open for creation, send reject signal to PES and go to step 4.

Step 2: For each of the security descriptors in the argument table do step 3.

Step 3: Find space for the descriptors in the SITM. If space is not available go to step 5. Move descriptor into space allocated. Update access vector by linking the descriptor in the list of discriptor s with the same high order S bits of identifier value (see Figure 38).

Step 4: Remove command from CSTM. Release argument space in CATM. Terminate.

Step 5: Send reject signal to the PES, release FIT, UIT MAUs allocated to file, and SITM space if any. Go to step 4.

ALGORITHM G: To process the command load-record.

Input Arguments: 1. CST entry
2. The pointer to CATM block where record is stored

Step 1: Check (from FIT) if file is open for creation. If file is not open for creation, then send reject signal to PES and go to step 8.

Step 2: Invoke algorithm A to determine the security atom to which the record belongs.

Step 3: Using the response from algorithm A, check against the security atom descriptors in the FIT of the file. If the response matches one of the descriptors in FIT, then retrieve the corresponding security atom number (see Figures 33 and 34). Call it AN and go to step 6.

Step 4: If the response of algorithm A does not match any of the descriptors in the FIT, then obtain control of the argument and response bus, and issue the request, "allocate security atom name for a file", with file name as the argument. Wait for SLIP response.

Step 5: If the SLIP response indicates that there are no more atom

names available, go to step 7, else, call the new atom name AN.

Step 6: Store AN in the CATM space (along with the record), set bit in CST entry to indicate that processing is completed and terminate.

Step 7: Send reject signal to PES.

Step 8: Release space in the CATM allocated for the record and other arguments of the command. Remove command from CSTM. Terminate.

ALGORITHM H: To process the command, Close-database-file.

Input Arguments: 1. CST entry.

Step 1: Check to see that the file is open. If it is already inactive, then send reject signal to PES and go to step 3.

Step 2: If file was open for access, then release space in SITM used for storing file sanctions. Also release space in UIT used for storing the file name and pointers. If this was the last file opened by the user, then release space occupied by the entire UIT.

Step 3: Remove command from CST and terminate.

ALGORITHM I: To process the command, open-database-file-for-access.

Input Arguments: 1. CST entry.
2. The pointer to file sanctions in CATM.

Step 1: Check if the file is already open for creation. If so, then send rejection signal to PES and go to step 7

Step 2: If type B protection is specified, then move file sanctions to SITM, and send file name and SITM address to SFP and then go to step 5.

Step 3: If type A protection is specified, then move file sanctions to the SITM.

Step 4: For each of the security atom descriptor found in the FIT of the file, invoke algorithm B to determine the access privileges accorded to the security atom. Form the atomic access privilege list, and store in the SITM.

Step 5: Enter file name, pointer to file sanctions and pointer to atomic access privilege list in the UIT.

Step 6: Turn on bit in FIT to indicate file is open for access.

Step 7: Remove command from CST, release space in CATM and terminate.

ALGORITHM J: To process the command, retrieve-by-query.

Input Arguments: 1. CST entry
2. The pointer to file sanctions in CATM

Step 1: Check if the file is open for access and also if the UIT contains the file name (see step 5 of algorithm I). If not, send reject signal to PES, remove the command from CST and terminate.

Step 2: Extract the default access descriptor for the user. Check if the user has a "retrieve-any" privilege (see Figure 11). If so, then go to step 5.

Step 3: If type B protection is specified, then go to step 6.

Step 4: If the type A protection is specified, then invoke algorithm C with the necessary arguments (i.e., access type = 'retrieve-by-query', conjunct=Q_1) for each of the conjuncts in Q.

Step 5: Store the response set S in the CATM and place a pointer in the CST.

Step 6: Turn on appropriate bit in the CST entry to indicate that processing of the command by CCRP is complete. Terminate.

ALGORITHM K: To process the command, retrieve-by-pointer.

Input Arguments: 1. CST entry
2. Pointer to CATM block where the record pointer is stored

Step 1: Check if the file is open for access and also if the UIT contains the file name (see step 5 of algorithm I). If not go to step 6.

Step 2: Extract the default access descriptor for the user. Check if the user has a "retrieve-any" privilege (see Figure 11). If so go to step 5.

Step 3: If type B protection is specified for the user, then go to step 5.

Step 4: If type A protection is specified, then look up atomic access privilege set for the user in the SITM. If the retrieve-by-pointer access is not allowed on the security atom specified in the pointer, then go to step 5.

Step 5: Turn on appropriate bit in the CST entry (see Figure 40) to indicate that processing of the command by CCRP is complete. Terminate.

Step 6: Send reject signal to PES, release CATM space and remove command from the CST. Terminate.

ALGORITHM L: To process the command, retrieve-by-query-with-pointers.

Input Arguments: 1. CST entry
2. The pointer to CATM where the query is stored

Step 1: Do steps 1-6 of algorithm J except in step 4, use access type= retrieve-by-query-with-pointer, and in step 2, check for "delete-any" privilege.

ALGORITHM M: To process the retrieve-within-bounds command.

Input Arguments: 1. Command status table entry
2. Pointer to argument table.

Step 1: Check if the file is open for access and if the UIT contains the file name. (Step 5 of algorithm I.) If not, then send reject signal to PES, remove command from CST and terminate.

Step 2: Check the default access associated with the database capability for the right to retrieve within bounds. If the right is not granted then go to step 4.

Step 3: Turn on the appropriate bit in the CST entry to indicate that processing of the command by CCRP is complete, terminate.

Step 4: Send reject signal to PES, release CATM space and remove command from CSTM. Terminate.

ALGORITHM N: To process the delete-by-query command.

Input Arguments: 1. CST entry
2. A pointer to the CATM where the query is stored.

Step 1: Execute steps 1 through 5 of algorithm J except in step 4, use access type = delete-by-query.

ALGORITHM O: To process the delete-by-pointer command.

Input Arguments: 1. CST entry
2. A pointer to CATM where the record pointer is stored.

Step 1: Execute steps 1 through 5 of algorithm K except in step 4 use access type = delete-by- pointer, and in step 2 check if user has "delete-any" privilege.

ALGORITHM P: To process the delete-file command.

Input Arguments: 1. CST entry
2. Default access descriptor.

Step 1: Check the CIT to see if the file is open for access (by some other user). If it is open then go to step 6.
Step 2: Check the argument default access descriptor to see if the user has the right to delete the entire file. If he does not have the right, go to step 6.
Step 3: For each attribute identifier in the FIT, issue the request delete-attribute-information to the SLIP.
Step 4: Release all MAUs allocated to the file, all attribute identifiers allocated to the file, all security descriptors in the SITM. Release space occupied by the FITM.
Step 5: Remove command from CSTM and terminate.
Step 6: Send reject signal to PES and go to step 5.

ALGORITHM Q: To process the insert-record command.

Input Arguments: 1. CST entry
2. A pointer to CATM where the record to be inserted is stored.

Step 1: Check if file is open for access and if the user information table contains the file name. If not, then send reject signal to PES, remove command from the CST and terminate.
Step 2: If type B protection is specified for the user go to step 13.
Step 3: Invoke algorithm A to determine the security atom descriptor for the record.
Step 4: Check the response from algorithm A against the security atom descriptors in the FIT then retrieve the corresponding atom number. Call it AN. If the response does not match, then go to step 9.
Step 5: Look up the atomic access privilege list to determine if the access "insert" is permitted on atom name AN. If it is not go to step 7.
Step 6: Store AN in the CATM (along with the record), set bit in CSTM entry to indicate that processing is completed. Terminate.
Step 7: Send reject signal to PES.
Step 8: Release space in CATM for the record and other arguments of the command. Remove command from CST. Terminate.

Step 9: [New security atom]. Invoke algorithm B to determine the access privilege accorded to the security atom by the file sanctions. Call the access privilege granted to the new atom, A.

Step 10: [Allocate a new security atom name]. Obtain control of the argument and response bus, and issue the request. "Allocate security atom name for a file" with a file name as argument to the SLIP. Wait for response from SLIP.

Step 11: If the SLIP response indicates that there are no more atom names available, go to step 7, else call it AN.

Step 12: Insert the pair (AN, A) into the access privilege list. Insert security atom descriptor into the FIT. Go to step 5.

Step 13: Send record for security check to the SFP. Wait for response from SFP.

Step 14: If SFP permits access "insert", then set bit in CST entry to indicate that the processing is completed. Terminate.

ALGORITHM R: To process the replace-record command.

Input Arguments:
1. CST entry
2. Two pointers one to the CATM block where the new record is stored and one to the CATM block where a pointer to the old record is stored.

Step 1: Check if file is open for access and if UIT contains the file name. If not, go to step 10.

Step 2: Look up default access descriptor (in the UIT). If user does not have "replace" access privilege, then go to step 10.

Step 3: If the type B protection is specified go to step 11.

Step 4: Extract the security atom name from the old record pointer associated with the command. Call it AN.

Step 5: Look up the atomic privilege list for AN. If the access "delete-by-pointer" is not permitted, then go to step 10.

Step 6: [Check if record may be inserted]. Execute steps 3, 4, 5, 6, 9, 10, 11 and 12 of algorithm Q to determine if insertion is permitted.

Step 7: If algorithm Q permits "insert", then set the bit in CST entry to indicate that processing is completed. Terminate.

Step 8: If algorithm Q does not permit "insert", then send reject signal to PES.

Step 9: Release space in CATM for the record and pointer argument of the command. Remove command from CST. Terminate.

Step 10: Send reject signal to PES and go to step 9.

Step 11: Send record to SFP for security check. Wait for response from SFP.

Step 12: If SFP permits "insert", then set the bit in CST entry to indicate processing by CCRP is complete. Terminate.

Notes: In order for a user to issue a "replace" command, he must have the privilege to "replace" a record on a file basis (see Figure 11). This is checked in step 2 above. Second, the file sanctions must grant him the privilege to delete the record to be replaced. Since

the user uses a pointer to indicate the record to be deleted, he must have the delete-by-pointer privilege on that record. For a user having type A protection, the delete-by-pointer access privilege must be granted on the security atom defined by the pointer. This is done in steps 4 and 5 above, Third, the file sanction must grant the user the privilege to insert the new record into the file. For a user having type A protection, the "insert-record" access privilege must be granted on the security atom defined by the record. This is done in a manner similar to the steps in algorithm Q.

ALGORITHM S: To process the command, retrieve-MAU-addresses.

Input Arguments: 1. CST entry

Step 1: Extract default access descriptor for the user on the file from the SITM. Check if the retrieve-MAU-addresses privilege is granted by the descriptor. If not, then send reject signal to PES, remove command from CST and terminate.

Step 2: Retrieve from FITM, the addresses of those MAUs that have been allocated to the file. Send the addresses to the PES. Remove command from CST and terminate.

ALGORITHM T: To process the command, load-creation-capability-list.

Input Arguments: 1. CST entry
2. A pointer to the CATM where user IDs are stored.

Step 1: Check to see if system ID matches with system ID stored in the SITM. If it does not, go to step 3.

Step 2: Store the user identifiers in the creation capability list in the FITM. Remove command from CST and terminate.

Step 3: Send reject signal to PES. Remove command from CST and terminate.

Note: This command is used to provide the DBC with a list of users who are authorized to create files in the DBC

2.4.3 Scheduling and Interrupt Handling

As the name suggests, this set of algorithms provides the CCRP the ability to control the sequence of events taking place in the DBCCP. Included in this set are the scheduler which constantly monitors the CST to determine the next command for execution, and interrupt handlers to handle interrupts from three sources - the SLIP, the SFP and PES. An interrupt from the PES has the highest priority; if the CCRP is executing a command processing algorithm or the scheduling algorithm, an interrupt from the PES will result in control being transferred to the PES interrupt handler (algorithm V below). However if the CCRP is executing an interrupt handling algorithm, no other interrupt will be honored until the interrupt handling algorithm has executed completely. An interrupt from SLIP will not be honored if the CCRP is executing a command processing or security related algorithm. The interrupt from SLIP must wait until the command processing algorithm has finished. There are two types of interrupts that can be raised by the SFP. Correspondingly, there are two interrupt handling algorithms in the CCRP. The data interrupt from SFP is raised when the SFP has response data to be sent to the PES. This interrupt has the same kind of priority enjoyed by the PES. The security interrupt from SFP is primarily intended to communicate security clearance/denial for certain types of commands from users having type B protection. This interrupt has the same priority as the SLIP interrupt.

In summary, the interrupt from PES and the data interrupt from SFP have the highest priority. The interrupt from SLIP and security interrupt have lower priorities. The priority of CCRP algorithms varies with respect to the CLIP, SFP, and the PES, depending on the type of algorithms being executed.

ALGORITHM U: To schedule the next event in the CCRP.

Input Arguments: 1. CST

Step 1: Check if an interrupt from SLIP is pending. If so give control to algorithm W.

Step 2: Check if a security interrupt from SFP is pending. If so give control to algorithm X.

Step 3: From the CST, pick up the first unexecuted command with the highest priority.

Step 4: Check if all arguments of the command has been received. If not skip the entry and choose the next one at the same priority level. If none exists, go to step 3 to obtain another command at the next level.

Step 5: Determine from command code, the processing algorithm to be executed. Turn appropriate bit on to indicate that CCRP processing of command has been initiated. Give control to the algorithm determined above.

ALGORITHM V: To process the interrupt from PES.

Input Arguments: None

Step 1: Obtain fixed part of the command (i.e., the command code, priority, user ID, file ID and number of arguments to follow) from PES.

Step 2: Allocate entry space in CST for the command and argument pointers. (See Figure 39).

Step 3: Enter the command information in the entry allocated in step 2. Link up command entry in the appropriate priority list.

Step 4: Determine if command is an access type command or a non-access type command. Turn on the appropriate bit if it is an access type command.

Step 5: If arguments of the command are available then for each argument, allocate storage in the CATM, and store the arguments. Place a pointer in the CST.

Step 6: Terminate.

Note: Arguments of a command may be given after a pause following the fixed part of the command. This implies that the above interrupt algorithm may be entered at step 5 instead of step 1.

ALGORITHM W: To process an interrupt from SLIP.

Input Argument: None

Step 1: Receive response data from SLIP over the argument and response bus.

Step 2: Identify by means of the request-ID and the CST entry, the restart address of the processing algorithm waiting for this response.

Step 3: Give control to the algorithm identified in step 2.

ALGORITHM X: To process the security interrupt from SFP.

Input Argument: None

Step 1: Receive security information from SFP over the communication bus.

Step 2: Identify by means of the request-ID and the command status table, the restart address of the processing algorithm waiting for this response.
Step 3: Give control to the algorithm identified in step 2.

ALGORITHM Y: To process the data interrupt from SFP.

Input Argument: None

Step 1: Receive data from SFP. Store data in database response memory. If buffer is full or data is exhausted, go to step 2.
Step 2: Transmit data in database response memory to PES. If more data is pending, go to step 1. Remove command entry from CST.

2.5 The Command Translation Processor (CTP)

The CTP is responsible for converting each of the access commands sent by the PES into a set of MAU orders. When a database file is being created or when a new record is to be inserted, the CTP has the additional responsibility of selecting the MAU in which the record will ultimately reside. In carrying out these functions, the CTP makes use of the data structures described earlier in Section 2.3. These data structures, which reside in the table memories accessed via the command argument-and-structure-loop-response bus, and the communication bus, are shared among the three processors of the DBCCP. In this section we describe the algorithms executed by the CTP. These algorithms may be divided, logically, into service algorithms, conversion algorithms and interrupt handling algorithms. Service algorithms provide the scheduling function, and the MAU selection function. Conversion algorithms translate access command into one or more MM orders. The CTP is capable of being interrupted by the SLIP and by the MM. Interrupt handling algorithms are designed to take care of these interrupts.

2.5.1 MAU Selection and Command Scheduling

ALGORITHM A: To select an MAU into which a record may be inserted.

Input Arguments:
1. Record to be inserted (in Argument table)
2. Mandatory clustering condition (MCC)
3. Set of optional clustering conditions (OCCs) and weights.

Step 1: [To obtain the MAUs whose records satisfy the MCC]. Let the number of conjuncts in the MCC be n, denote the i-th conjunct of MCC by Q_i. Set $i \leftarrow 1$; Set $S \leftarrow \phi$.
Step 2: Obtain control of the argument and response bus in preparation for issuing a request to the structure loop interface processor.
Step 3: Issue the request, "Retrieve the MAU addresses for the query Q_i" to the SLIP with arguments request-identification, file identification.

Step 4: Wait for response set from SLIP. [At this point control goes back to the scheduling algorithm. When an interrupt occurs, control is returned to step 5].

Step 5: Call the response set S'. S←SUS'. i←i+1. If $i \leq n$, then go to step 2.

Step 6: If S is empty, then go to step 18.

Step 7: Define $\omega \equiv \{(f,W) \mid f \in S$ and W is a constant equal to 0$\}$

Step 8: [To select one MAU out of the set S]. Let there be m optional clustering conditions. The j-th OCC has K_j conjuncts. The p-th conjunct of the j-th OCC will be denoted by Q_j^p. The cluster weight of the j-th OCC is denoted by C_j. Set j←1, p←1, $T_j \leftarrow \phi$.

Step 9: Obtain control of the argument and response bus in preparation for issuing a request to the SLIP.

Step 10: Issue the request, "Retrieve the MAU addresses for the query Q_j^p" to the SLIP with the arguments: request ID, and file identification.

Step 11: Wait for response set from SLIP. [At this point control goes back to the scheduling algorithm. When an interrupt occurs, control is returned to step 12].

Step 12: Call the response set T_j^p. $T_j \leftarrow T_j \cup T_j^p$. p←p+1. If $p \leq K_j$ then go to step 9.

Step 13: For each MAU address f in T_j do step 14.

Step 14: If $f \in S$, then replace (f,W) in ω by $(f,W+C_j)$.

Step 15: j←j+1. If $j \leq m$, then set p←1, $T_j \leftarrow \phi$ and go to step 9.

Step 16: From the set ω, select the couple(s) $\{(f_i,W_i)\}$ such that $W_i \geq W_\ell$ for all $\ell \leq N$ where N is the cardinality of ω.

Step 17: Extract the MAU addresses from the couples selected in step 16. Call the set of such MAU addresses θ. Terminate.

Step 18: [No MAU has at least one record to satisfy the MCC]. Set $\theta \leftarrow \phi$. Terminate.

Response: θ contains the set of MAUs found eligible to contain the record.

ALGORITHM B: To initiate the translation of an access command.

Input Arguments: 1. CST

Step 1: Check if an interrupt from SLIP is pending. If so give control to algorithm H.

Step 2: Check if an interrupt from MM is pending. If so give control to algorithm I.

Step 3: Extract the status information of the first command in the highest nonempty priority list.

Step 4: Check if the command is an access command or a non access command.

Step 5: If the command is an access command go to step 7, else go to step 6.

Step 6: Extract the status of the next command at the same priority. If no command at the same priority exists go to step 9, else go to step 4.

Step 7: If the command is being processed by CCRP and processing is not complete then go to step 6.

Step 8: [Access command that can be translated is found]. Determine from the command code the access algorithm to be executed.

Turn on appropriate bit in the status information table entry to indicate that the CTP processing of the command has been initiated. Give control to the algorithm determined above. [All processing algorithms return control to step 1 above].

Step 9: Extract the first command in the next highest non-empty priority list and go to step 4. If no such list is available go to step 1.

Response: None

2.5.2 Command Translation

There are five algorithms under this category which handle the eight access commands listed in Table I in Section 2.3. Two counters known as the database object counter and the order number counter are maintained by the CTP. The first is used to uniquely identify an argument of an MM order. The second counter is used to identify the order itself.

ALGORITHM C: To process a load-record or insert-record access command

Input Arguments: 1. CST entry

Step 1: From the CST entry, extract the pointer to the mandatory clustering condition (in the CATM) and the pointers to the optional clustering conditions.

Step 2: Invoke algorithm A with the pointer retrieved in step 1 and the file name as the arguments.

Step 3: [The set θ contains the response of algorithm A] If the set θ is empty, then go to step 17.

Step 4: For each member f of θ do step 5.

Step 5: Look up MAU space allocation table for f. Extract amount of space available in the f-th MAU. Call it b_f.

Step 6: Define the set $\theta' \equiv \{(f,b_f) \mid f \in \theta \text{ and } b_f \geq \text{record size}\}$. If θ' is empty go to step 17.

Step 7: From the set θ' choose the MAU f such that the corresponding value of b_f is smallest. If more than one such MAU should be found, make an aribitrary choice (say the first one) among the MAUs. Call this choice f_s.

Step 8: [Determine cluster number for recrod]. For the set of cluster keywords occurring in the record, issue the request "Retrieve cluster identifier with count", to the SLIP with the following arguments: a) file name, b) request identifier (obtained from the command status table) and c) pointer to CATM block where the cluster keywords are stored.

Step 9: Wait for SLIP response. [At this point control goes back to the scheduling algorithm. When the SLIP response is available, control is given to step 10 via the interrupt handler].

Step 10: If the response is empty, then go to step 11, else go to step 14.

Step 11: Issue the request, "Allocate cluster identifier for a file" to SLIP with the following arguments: a) file name, b) request identifier (obtained from the CST entry).

Step 12: Wait for SLIP response. [At this time control is given back to the scheduling algorithm. When the SLIP response is available, control comes back to step 13].

Step 13: If the response is empty go to step 22.

Step 14: Call the cluster identifier in the SLIP response set c.

Step 15: Obtain the database object counter value K. Obtain the order number counter value. Concatenate it with the command identifier in the command status table. Call the concatenation N. Increment both counters. Issue the MM order, "Insert Record" with the following arguments: a) database object number K, b) order number N, c) the MAU address f_s and d) the database object consisting of the record and the triple <MAU address f_s, cluster identifier c, security atom name sa>. [The MAU address is determined in step 7 or 21 and the cluster identifier is determined in step 14. The security atom identifier (if Type A protection is specified) is stored in the argument table by CCRP.

Step 16: Update the MAU space allocation table entry for f_s by the number of bytes occupied by the record. Go to step 24.

Step 17: Extract the set of addresses of MAUs allocated to the file from FIT. Call this set θ.

Step 18: From the MAU allocation table, obtain the space available for each of the MAU address in θ. Call the space available in the MAU 'f' b_f.

Step 19: Form the set $\theta' \equiv \{(f,b_f) \mid f \in \theta, b_f \geq \text{record size}\}$. If θ' is empty go to step 20, else go to step 7.

Step 20: [Allocate new MAU]. Search the MAU bit map for a free MAU. If no MAU is found go to step 22.

Step 21: Call the MAU allocated in step 20 f_s. Go to step 8.

Step 22: Reject the command by sending a reject signal to PES.

Step 23: Release space occupied by the record and clustering conditions in the CATM and remove command from CST. Terminate.

Step 24: [Update SM]. For each nonclustering keyword in the record, issue the request, "Insert index term" to SLIP with the following arguments: a) index term (f_s,c,sa), b) transformed keyword value T(K) available in the record.

Step 25: For each clustering keyword in the record, issue the request, "Insert index term" to SLIP with the following argument: a) index term $((f_s,c,sa),q)$ where q is the number of clustering keywords in the record, b) transformed keyword value T(K) available in the record.

Step 26: Set appropriate bit in CST entry to indicate processing of command by CTP is completed. Release space occupied by record in CATM. Terminate.

ALGORITHM D: To process a retrieve-by-query-with-pointer retrieve-by-query or delete-by-query access command.

Input Arguments: 1. Pointer to entry in command status table.

Step 1: From the entry in the CST, extract the pointer to the CATM where the query is stored.

Step 2: Let there be m query conjuncts. For each of the query conjunct Q_j do steps 3 through 8.

Step 3: Issue the request, "retrieve MAU addresses" to the SLIP with the following arguments: a) file name, b) request identification (from command identifier in status table), c) pointer to CATM where query is stored.

Step 4: Wait for SLIP response. [At this point control goes back to the scheduling algorithm. When the SLIP response is available, control is given back to step 5 via the interrupt handler.]

Step 5: Obtain the database object counter value K.

Step 6: For each of the MAU addresses in the SLIP response, do step 7.

Step 7: Concatenate the command identifier (from the command status table) with an order number read off the order number counter. Call it N. Increment the order number counter. Issue the MM order, "Retrieve/Delete-by-query or retrieve-by-query-with-pointer" with the following arguments: a) database object number K, b) order number N, c) the MAU address f and d) the database object (Q_j, set of security atoms names if specified by CCRP).

Step 8: Increment database object counter.

Step 9: Release space occupied by the query, and set the appropriate bit in the CST entry to indicate that processing by CTP is completed. Terminate.

ALGORITHM E: To process a retrieve-by-pointer or delete-by-pointer access command.

Input Arguments: 1. CST entry

Step 1: From the entry in the CST entract the pointer to the CATM where the record pointer is stored.

Step 2: From the pointer extract the MAU address f. Obtain the database object counter value K. Concatenate the command identifier with the order number read off the order number counter, call it N. Increment both counters. Issue the MM order, "Retrieve/delete by pointer" with the following arguments: a) MAU address f, b) database object number K, c) order number N, d) pointer consisting of the triple <record identifier, cluster identifier, security atom number>.

Step 3: Release space occupied by the pointer in the CATM, and set the appropriate bit to indicate that CTP is completed. Terminate.

ALGORITHM F: To process a replace-record access command.

Input Arguments: 1. CST entry.

Step 1: From the entry in the command status table, extract the pointer to the argument table where the record to be inserted and the pointer to the old record are stored.

Step 2: [Insertion of new record]. For the set of cluster keywords occuring in the new record, issue the request, "Retrieve cluster identifier with count", to the SLIP with the following arguments: a) file name, b) request identifier (from command status table) and c) pointer to argument table where the cluster keywords are stored.

Step 3: Wait for SLIP response. [At this time point, control goes back to the scheduling algorithm. When SLIP response is available, control is given to step 4 via the interrupt handler].

Step 4: If the response is empty then go to step 5, else go to step 8.

Step 5: Issue the request, "Allocate cluster identifier for a file" to SLIP with the following arguments: a) filename, b) request identifier (from command status table).

Step 6: Wait for SLIP response. [At this time control is given back to the scheduling algorithm. When the SLIP response is available, control comes back to step 7].

Step 7: If the response is empty go to step 16.

Step 8: Call the cluster identifier in the SLIP response set c.

Step 9: Obtain the database object counter value K. Obtain the order number counter value. Concatenate it with the command status table. Call the concatenation N. Increment both counters. Issue the MM order, "Insert Record" with the following arguments: a) database object number K, b) order number N, c) the MAU address f extracted in step 7, d) the database object consisting of the record and the triple <MAU address f, cluster c, security atom AN>. The cluster c is determined in step 8. The atom number AN is determined by CCRP. (See algorithm Q under CCRP).

Step 10: Update the MAU space allocation table entry for f by the number of bytes occupied by the record.

Step 11: [Update SM]. For each nonclustering keyword in the record, issue the request, "Insert index term" to the SLIP with the following arguments: a) index term (f,c,AN), b) transformed keyword T(K) available in record.

Step 12: For each clustering keyword in the record, issue the request, "Insert index term" to SLIP with the following arguments: a) index term ((f,c,AN),q) where q is the number of clustering keywords in the record, b) transformed clustering keyword value T(K) available in the record.

Step 13: [Delete old record]. Extract pointer to the old record from the argument table.

Step 14: From the pointer extract the MAU address f. Obtain the database object counter value K. Concatenate the command identifier with the order number counter value. Call it N. Increment both counters. Issue the MM order, "Delete-by-pointer" with the following arguments: a) MAU address f, b) database object number K, c) order number N, d) pointer consisting of the triple <record identifier, cluster identifier, security atom number>.

Step 15: Release space in CATM occupied by the new record and pointer to old record. Set appropriate bit in CST entry to indicate that processing by CTP is complete. Terminate.

Step 16: Send reject signal to PES. Release space in CATM occupied by new record and pointer to old record and delete the command from CST. Terminate.

Notes: The replace command is translated into 2 MM orders by the above algorithm. First, the new record is inserted, and then the old record is deleted. Steps 2 through 12 are responsible for the insertion process, while steps 13 and 14 generate the delete order. The insertion process is similar to the processing

for an "Insert Record" command except that no clustering is attempted here. The rationale for this is that the user usually wants the new record to be located in the same MAU as the old one. The deletion process is the same as the processing for a delete-by-pointer command.

ALGORITHM G: To process a retrieve-within-bounds access command.

Input Arguments: 1. CST entry

Step 1: From the entry in the command status table, extract the pointer to the argument table.

Step 2: Extract from the argument table the lower bound and upper bound of the set of records to be retrieved. [The bounds are actually pointers in the format shown in Figure 13].

Step 3: Check if the MAU addresses specified in the two bounds are the same. If they are not, go to step 7.

Step 4: Extract record identifiers from the two bounds. Call them R1 (lower bound) and R2 (upper bound).

Step 5: Obtain the database object counter value K. Obtain the order number counter values. Concatenate it with the command identifier in the command status table. Call the concatenation N. Increment both counters. Issue the MM order, "Retrieve within bounds", with the following arguments: a) database object number K, b) order number N, c) the MAU address extracted in step 3 and d) R1 and R2.

Step 6: Release space occupied by the pointers in the CATM. Set appropriate bit in CST entry to indicate that processing by CTP is completed. Terminate.

Step 7: Send reject signal to PES. Release space occupied by the pointers in the CATM. Remove command from CST. Terminate.

2.5.3 Interrupt Handling in the CTP.

The algorithms in this section are executed when interrupts from the SLIP or the MM are to be serviced. The CTP can be interrupted by two sources - the SLIP and the MM. The SLIP interrupts the CTP when it has received structural information from the structure loop. This information must have been previously requested by a processing algorithm (see Section 2.5.2 above). The MM requests an interrupt a) to indicate non-acceptance of an order due to a security violation (type B protection), b) to transmit MAU space information and c) to transmit update information to be sent to the SM during a compaction operation (see discussion under section 3.6). An interrupt from either the SLIP or the MM is accepted by the CTP only when the CTP is executing step 1 or step 2 of the scheduling algorithm (see algorithm B above). When the CTP is executing a processing algorithm or servicing an interrupt or when the CTP is scheduling the execution of a new command from the status table, all interrupts are masked off.

ALGORITHM H: To process an interrupt from the SLIP.

Input Argument: None

Step 1: Receive response data from SLIP over argument and response bus.
Step 2: Identify by means of the request identifier and the command status table, the restart address of the processing algorithm waiting for this response.
Step 3: Give control to the algorithm address identified in step 2.

ALGORITHM I: To process an interrupt from the MM.

Input Arguments: 1. Cause of interrupt.
2. Order number.

Step 1: If cause of interrupt is security violation, go to step 4.
Step 2: If cause of interrupt is to update MAU space allocation table, go to step 7.
Step 3: If cause of interrupt is update information, go to step 5.
Step 4: [Security violation]. From order number, extract command identifier. With the help of the command ID, locate the entry in CST for the command. Increment the count of security violation in the entry. Terminate.
Step 5: [Update SM]. For each 4-tuple of the form <Transformed keyword, MAU address, cluster number, security atom name>, issue the SLIP request, "Delete an index term" with the above 4-triple as the argument.
Step 6: Receive (from MM) amount of space available in the MAU. Update the MAU space allocation table accordingly. Terminate.
Step 7: [Update MAU space allocation table]. Receive data pertaining to maximum space available on a track of an MAU. Store this information in the appropriate field of the entry in the MAU space table [see Figure 36]. Terminate.

2.6 The Structure Loop Interface Processor (SLIP)

Unlike the CCRP and the CTP which manipulate data structures described in Section 2.3, the SLIP maintains local data structures in order to carry out its functions. In the following discussion, we first examine the data structures in the SLIP, and then present the algorithms executed by the SLIP. Finally we propose a hardware organization to realize it.

2.6.1 Data Structures in the SLIP

Service requests generated by the CCRP and CTP are placed in queues known as the request queues. Certain requests which have higher priority than others must be executed before requests of lower priority are executed. The status of a request after it has been placed in the queue must be maintained by the SLIP. When a request has been serviced, the SLIP must cause an interrupt to the appropriate processor (CCRP or CTP). The result of a request must be made available on the command-argument-and-structure-loop-response bus when the CCRP or CTP is ready to receive the

information. With this discussion as the background, we are ready to examine the main data structure maintained by the SLIP, the request status table (RST).

As shown in Figure 42a,the RST has two parts: a list of pointers one for each priority level, and a table of status information entries, one for each request. The entries in the table at the same priority level are linked together. Requests within a priority level are treated on a FIFO basis. The entry for the first request at a priority level is pointed to by the appropriate entry in the list headers block. The format of an entry in the table of status information is shown in Figure 42b. There are eight fields in an entry. The first field identifies the file referenced by the request. The second field identifies the request itself. Since several requests may be in various stages of completion at various components in the structure loop, it is important to tag each request in this way in order that the response from the loop can be ultimately paired with the corresponding request. [The request ID is generally the command ID in the CST described in Section 2.3. The CCRP or CTP merely transmits the command ID at the time of requesting service from the SLIP]. The request source field indicates which processor (CTP or CCRP) placed the request. The request code identifies the service desired by the requestor. A list of possible request codes and their explanation is given in Table III. The status field indicates the progress made by a request towards completion. More specifically, the status bits indicate whether a request has been initiated or not, whether it has been completed or not, error codes from the structure loop components.etc. Each of the requests can have a variable length arguments. These arguments are stored in the CATM described earlier. A pointer to the argument area is stored in the sixth field. The seventh field contains a pointer to the result buffer where the structure information requested by the CTP or CCRP is stored. The last field points to an entry of a request at the same priority level and which arrived next in time sequence. The number of entries in the RST is a design parameter which is to be determined on the basis of the anticipated processing speeds of the structure loop components and the CRT and CCRP.

2.6.2 The SLIP Logic

The logic of SLIP consists of algorithms which are executed a) in response to requests placed in the request queues, b) in response to interrupts generated by the structure loop components (principally by the

Table III. Request Types Accepted by SLIP

Code (octal)	Request Type
001	Retrieve MAU addresses
002	Retrieve cluster identifier with count
003	Retrieve security atom names
004	Retrieve MAU address, cluster idenitfier and security atoms
005	Insert index term
006	Delete index term
007	Load attribute information for an attribute
010	Load hash algorithms for a file
011	Allocate MAU number for a file
012	Allocate cluster identifier for a file
013	Allocate security atom name for a file
014	Deallocate MAU number for a file
015	Deallocate cluster identifier for a file
016	Deallocate security atom name for a file
017	Translate MAU address into MAU number
020	Delete attribute information for file
021	Deallocate all MAU numbers, cluster identifiers and security atom names for a file

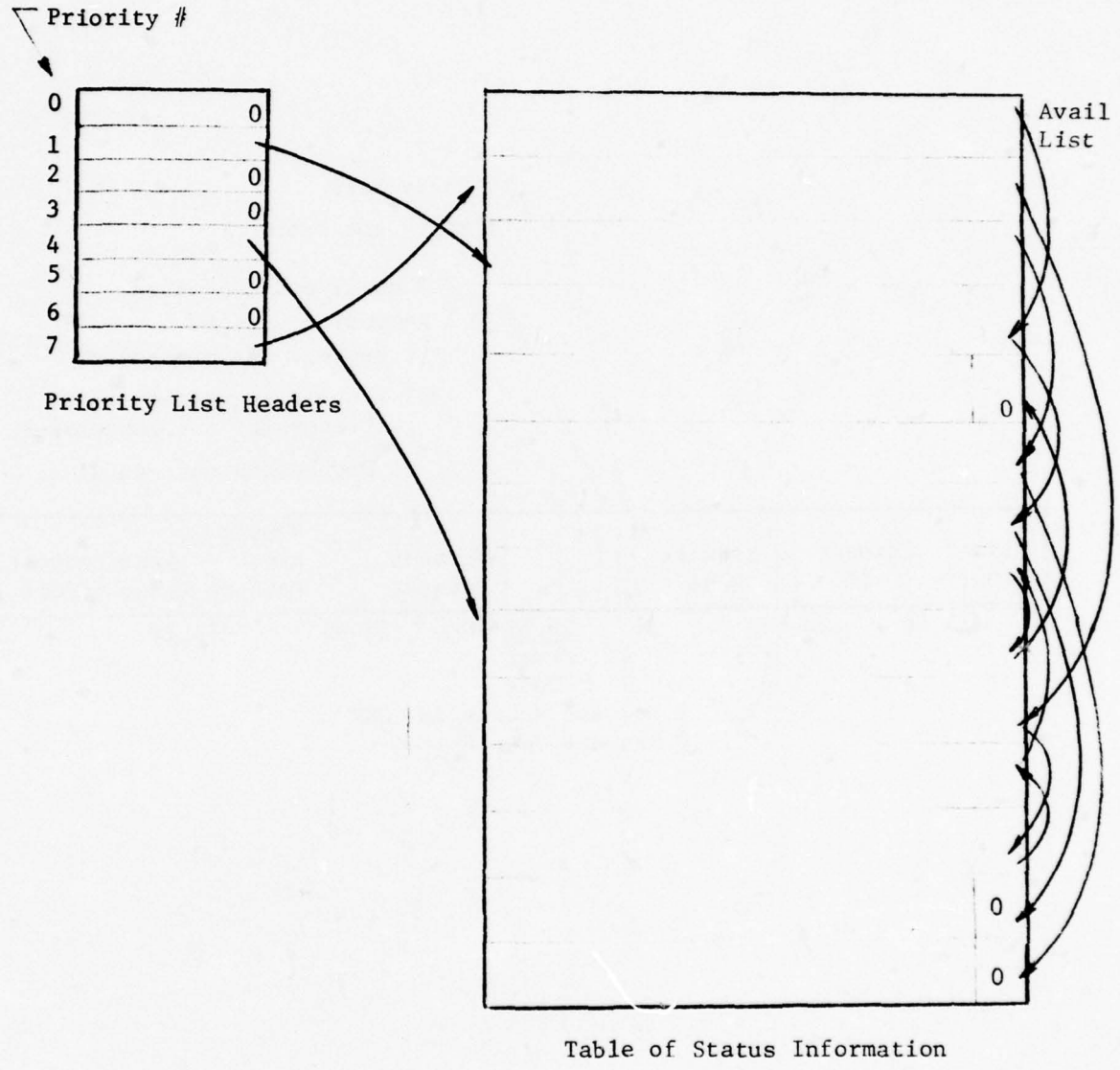


Figure 42a. Format of Request Status Table (RST)

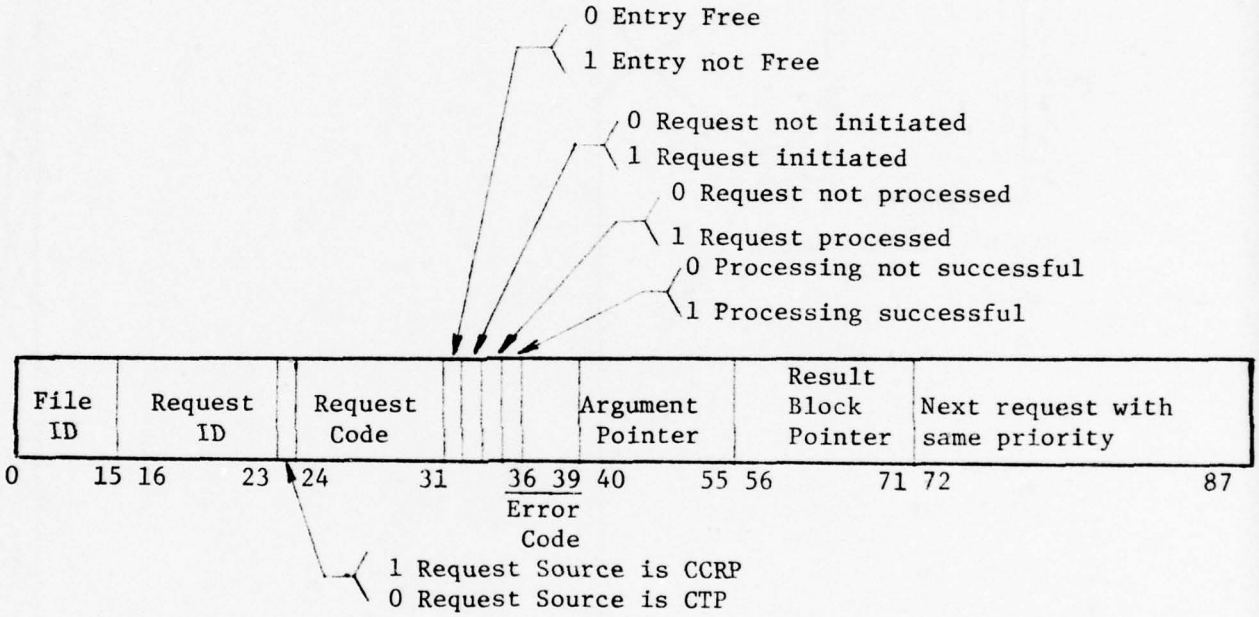


Figure 42b. An Entry in the table of status information

IXU), c) for the maintenance of request queues and d) for communicating with the CTP and CCRP over the command-argument-and-structure-loop-response bus. We, therefore, classify these algorithms as request-initiating, interrupt-driven and service algorithms.

A. Request Initiation - Under this category, there is one algorithm for each of the requests identified in Table III.

ALGORITHM A: To retrieve MAU addresses

Input Arguments:
1. The file name F-ID from RST entry
2. The conjunct of keyword predicator of the from $T(K_1) \wedge T(K_2) \wedge T(K_3) \wedge \ldots \wedge T(K_n)$ from CATM
3. The request identifier R-ID from RST entry

Step 1: Issue to the IXU, the command, "extract-from-the-next-set-of-index-terms-(whose request identifier is R-ID and which belong to file F-ID)-the-MAU-address".

Step 2: For every keyword predicate in the argument, conjunct do steps 3 and 4.

Step 3: Issue to the SM the command, retrieve-the-index-terms-of-the-keywords-which-satisfy-the-predicate-$T(K_i)$. [The predicate type ('=', '≠', '≤', '<', '>', '≥') and request ID are sent as arguments of the command]

Step 4: Issue to the KXU the command, transform-the-keyword-K_1-belonging-to-file-F-ID-into-its-encoded-form. Place the transformed value in bits 8-55 of predicate.

Step 5: Issue to the SM, the command, "Reset",. [This command indicates the completion of a set of "retrieve" commands given in step 3.

Step 6: Set status bits to indicate that the request has been initiated. Terminate.

ALGORITHM B: To retrieve cluster identifier with count.

Input Arguments:
1. The file name F-ID from RST
2. The conjunct of keywords with count q from CATM
3. The request identifier R-ID from RST

Step 1: Issue to the IXU, the command, extract-from-the-next-set-of-index-terms-(whose request identifier is R-ID and which belong to the file F-ID)-the-cluster-identifiers.

Step 2: For every keyword in the argument conjunct do steps 3 and 4.

Step 3: Issue to the SM the command, retrieve-with-count-the-index-terms-of-the-keyword (whose transformed value is to be obtained from KXU). [The request-ID is sent as an argument].

Step 4: Issue to the KXU the command, transform-the-keyword-K_1-belonging-to-file-F-ID-into-its-encoded-form. Place transformed value in bits 8-55 of keyword.

Step 5: Set status bits in RST (see Figure 42b) to indicate that the request has initiated. Terminate.

ALGORITHM C: To retrieve security atom names.

Input Arguments:
1. The file name F-ID
2. The conjunct of keyword predicates
3. The request identifier R-ID

Step 1: Issue to the IXU, the command, extract-from-the-next-set-of-index-terms-(whose request identifier is R-ID and which belong to the file F-ID) the-security-atom-name.
Step 2: For every keyword predicate in the argument conjunct do steps 3 and 4;
Step 3: Issue to SM the command, retrieve-the-index-terms-of-the-keywords-which-satisfy-the-predicate-T(K_i). [The predicate type ('=', '≠', '<', '≤', '>', '≥'), and request identifier are sent as arguments of the command].
Step 4: Issue to the KXU the command, transform-the-keyword-K_1-belonging-to-file F-ID-into-its encoded form. Place transformed value in bits 8-55 of predicate.
Step 5: Issue to the SM the commamd 'reset'.
Step 6: Set status bits in the RST (see Figure 42b) to indicate that the request has been initiated. Terminate.

ALGORITHM D: To retrieve MAU addresses, cluster identifier and security atom names.

Input Arguments:
1. The file name F-ID
2. The conjunct if keyword
3. The request identifier R-ID

Step 1: Issue to IXU the command, extract-from-the-next-set-of-index-terms (whose request identifier is R-ID and which belongs to the file F-ID) the MAU addresses the cluster identifiers and security atom names.
Step 2: Execute steps 2-6 of algorithm C.
Step 3: Terminate.

ALGORITHM E: To insert an index term for a keyword.

Input Arguments:
1. The index term i
2. The count (q) if keyword is clustering keyword
3. The transformed value of keyword T(K)

Step 1: If argument keyword is security/clustering keyword then go to step 3; else go to step 2.
Step 2: Issue to the SM the command, insert-index-term-i-for-the-keyword-whose-transformed-value-is-T(K). Go to step 4.
Step 3: Issue to the SM the command, insert-index-term-(i,q)-for-the-keyword-whose-transformed-value-is-T(K).
Step 4: Set status bits in the RST (see Figure 42b) to indicate that the request has been initiated, and processing is completed successfully.
Step 5: Terminate.

ALGORITHM F: To delete an index term.

Input Arguments:
1. The index term i
2. The transformed value of keyword T(K).

Step 1: Issue to the SM the command, delete-index-term-i-for-the-keyword-whose-transformed-value-is-T(K).
Step 2: Set status bits in the RST (see Figure 42b) to indicate that the request has been initiated, and processing is completed successfully.
Step 3: Terminate.

ALGORITHM G: To load attribute information for an attribute.

Input Arguments: 1. The attribute identifier A-ID.
2. The attribute information in the CATM.

Step 1: Issue to the KXU the command, create-an-AIT-block-for-the-attribute-identifier-A-ID. [The attribute information is sent as argument to this command].
Step 2: Set status bits in the RST to indicate that the request has been initiated, and completed successful.
Step 3: Terminate.

ALGORITHM H: To load hash algorithms for a file.

Input Arguments: 1. The file name F-ID
2. The number of hash algorithms 'k'
3. The hash algorithms

Step 1: Issue the command to the KXU, build-a-set-of-k-hash-algorithms-for-the-file F-ID.
Step 2: Transmit the k hash algorithm to the KXU.
Step 3: Set status bits in RST to indicate that the request has been initiated, and processed successfully.
Step 4: Terminate.

Notes: In algorithm G and H, if the KXU rejects the input on account of lack of table space, then the status of the request is set to unsuccessful completion (bit 35 is set to 0) and the error code is set to indicate cause of rejection.

ALGORITHM I: To process an allocate/deallocate request to IXU.

Input Arguments: 1. The file name F-ID
2. The MAU number, cluster identifier or security atom number in case of a deallocate request
3. The MAU address M_f in case of an MAU number request.

Step 1: For a deallocate request, issue the command, deallocate-for-file F-ID-the-MAU-number-cluster-identifier-or-security-atom-name-specified-by-the-second-argument. Go to step 3.
Step 2: For an allocate-MAU-number-request, issue the command allocate-an-MAU-number-for-the-absolute-MAU-address-M_f-and-file-F-ID. For an allocate-cluster-identifier/security-atom-name-request, issue the command, allocate-cluster-identifier/security-atom name.
Step 3: Set status bits in the RST to indicate request has been initiated. Terminate.

ALGORITHM J: To translate MAU address into the corresponding MAU number.

Input Arguments: 1. The file name F-ID
2. The MAU address M_f

Step 1: Issue the command to the IXU, convert-MAU-address into Mau number. Send F-ID and M_f as arguments.
Step 2: Set status bits in RST to indicate that the request has been initiated. Terminate.

ALGORITHM K: To delete attribute information for a file.

Input Arguments: 1. The file name F-ID
2. Number of attributes m
3. The m attribute identifiers

Step 1: Issue the command to the KXU, delete-AITs-and hash-algorithms-of-a-file, and send F-ID and m attribute identifiers to KXU as arguments.
Step 2: Set status bit in RST to indicate that the request has been initiated and processed successfully. Terminate.

ALGORITHM L: To deallocate all Mau numbers, cluster identifiers and security atom names for a file.

Input Arguments: 1. The file name F-ID

Step 1: Issue the command to the IXU, deallocate-all-index-translation-information, and send file name F-ID as an argument.
Step 2: Set status bits in RST to indicate that the request has been initiated and processed successfully. Terminate.

B. Interrupt Handling - Under this category are algorithms executed in response to interrupts generated by the IXU.

ALGORITHM A: To process the IXU interrupt for transmission of retrieved MAU addresses, cluster idevlifian security atom names or triples of the form <MAU addresses, cluster identifier, security atom names>.

Input Arguments from IXU (source of interrupt):
1. The request identifier R-ID
2. The number n of retrieved items
3. The n items

Step 1: Locate the entry in the RST for the request R-ID.
Step 2: Allocate in the result buffer enough space to store n of the retrieved items. If there is no space, then go to step 5.
Step 3: Place a pointer to the memory (allocated in step 2) in the RST entry. Receive the items from IXU, and place them in the result buffer.
Step 4: Set status bits in RST to indicate successful completion of the request. Terminate.
Step 5: [No space in result buffer]. Reject output from IXU, place request at the beginning of the queue and terminate.

ALGORITHM B: To process the IXU interrupt from transmitting allocation information.

Input Arguments from IXU:
1. The request identifier R-ID
2. The MAU number/cluster identifier/security atom name

Step 1: Locate the entry in the RST for the request R-ID.
Step 2: Receive the MAU number/cluster identifier/security atom name from IXU and place it in the result field (field 7 in Figure 42b) of the RST entry located in step 1.
Step 3: Set status bits in RST to indicate successful completion of the request. Terminate.

C. Service Algorithms - Under this category, there are algorithms which place a request in the RST, remove a request from the RST allocate space in the result buffer.

ALGORITHM A: To place a request in the RST

Input Arguments:
1. The file name F-ID
2. The request identifier R-ID
3. The request code
4. A pointer to the arguments in CATM
5. The priority of the request

Step 1: Allocate the first entry in the list of available entries (see Figure 42a) in the RST for the current request. If the list is empty, go to step 4.

Step 2: Place file name, request identifier, request code and argument pointer in their respective fields. Clear status field. Set the status bit to indicate entry is occupied.
Step 3: Link the entry into the appropriate list of priority. Terminate.
Step 4: [RST is full]. Reject request. Terminate.

ALGORITHM B: To locate and remove requests that have been processed by the struction loop.

Input Argument: None

Step 1: For each priority list in the RST do steps 2 through 4.
Step 2: Scan the list of requests for the i-th priority level. For each request whose status indicates completion do step 3 through 4.
Step 3: Determine the response for the request from the result field or result buffer and send it to the CCRP or CTP (depending upon the origin of the request.)
Step 4: Set the status bits in the entry to indicate that the entry is available for allocation. Link the entry into the list of available entries. Release result buffer space by invoking algorithm C.
Step 5: Terminate.

Note: This algorithm is continually executed, so that as soon as requests are processed, their responses can be sent to their respective sources (CTP or CCRP).

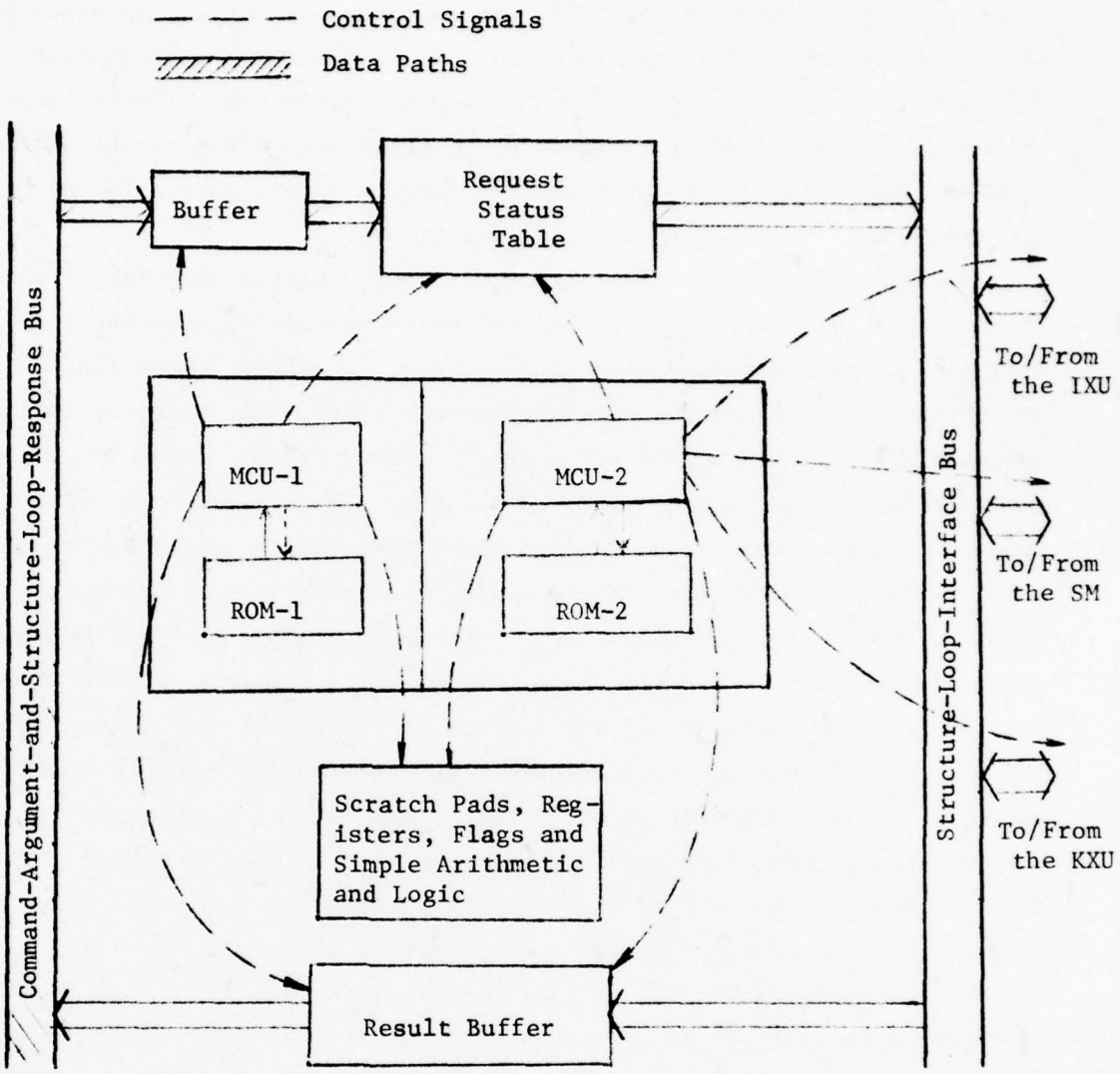
ALGORITHM C: To allocate or deallocate memory in result buffer as and when requested, one block at a time.

Input Argument: This address of the memory block if deallocation is desired, or the size of memory if allocation is desired.

Step 1: If deallocation is requested go to step 4.
Step 2: [First Fit]. Determine the first block in the chain of available blocks, whose size is large enough for the request. If no such block is found, go to step 5.
Step 3: Transmit address determined in step 2 to requestor. Terminate.
Step 4: [Deallocation]. Link block at the end of the list of available blocks. Terminate.
Step 5: [No space in result buffer]. Reject request. Terminate.

2.6.3 Hardware Organization of the SLIP

From the above discussions on the data structures and the algorithms executed by the SLIP, we find that there is a need for concurrency of execution of service algorithms on one hand and interrupt and structure loop initiating algorithms on the other hand. Requests which are made by the CCRP and CTP over the command-argument-and-structure-loop-response bus are placed in the RST by the service micorcontrol unit (MCU-1) (See Figure 43).



Notes: Scratch Pad Registers and ALU are shared by MCU-1 and MCU-2.
MCU-1 is a Micro control unit for executing service algorithms.
MCU-2 is a Micro control unit for processing interrupts and executing loop initiating algorithms.
MCU-1 and MCU-2 have the micro instruction address generation logic and operate concurrently.
ROM-1 contains micro instruction sequences for all service algorithms.
ROM-2 contains micro instruction sequences for all interrupt and loop initiating algorithms.

Figure 43. Hardware Organization of Structure Loop Interface Processor

while requests already in the table are initiated by MCU-2. Accesses to the RST by the two control units are serialized by means of hardware locks. Information from the structure loop is stored in the result buffer under the control of MCU2, while information is moved out of the result buffer by MCU1. Here again hardware locks are used to ensure that only one unit has access to the buffer at any given time.

The RST and result buffers are small random access memories of sizes in the range 16-32K bytes. The control units are microprogrammed to incorporate the algorithms discussed earlier. The ROMs (ROM-1 and ROM-2) have one program for each of the algorithms. The microcontrollers generate the appropriate ROM addresses to execute them. MCU-2 has three interrupt lines one each for IXU, SM and KXU. The IXU uses its interrupt line to indicate that it has decoded index terms for a particular request. The KXU uses the interrupt line to indicate that it is ready to accept the next keyword for transformation. The SM interrupts only to indicate an error condition. In addition to the interrupt lines, the MCU-2 has control lines by which it can monitor the activities of the IXU, SM and KXU and grant access to the structure-loop-interface bus (SLIB). MCU-2 can also initiate transfer of arguments for various commands from the command-argument-and-structure-loop response bus (CASRB) to the SLIB. The MCU-1 responds to requests on the CASRB by queuing the requests in the RST. It is also responsible for transferring data out of the result buffer and sending it to the proper requestor via the CASRB. The two microcontrollers share all of the logic capabilities of the SLIP. As shown in Figure 43, the two controllers can gain access to (on a specialized basis) the scratch pad registers, and a simple arithmetic and logic unit.

3. THE MASS MEMORY (MM)

The mass memory (MM) is the repository of the database. Contemporary database management systems also store their databases in mass memories (although the MM is conceptually and physically different from contemporary mass memories). In addition, most of the systems store meta-information about their databases in mass memories. Examples of such information are pointers, indexes, and security related information. In order to access a piece of data in these systems, one must first access the meta-information. It is, therefore, likely that such organizations can lead to an excessive number of accesses to a relatively slow mass memory. A related problem in software-laden systems is that the load on the central processor tends to increase substantially as the number of accesses increases. This is because each access to a mass memory is preceded by several tens of processor instructions required to prepare and issue an access command.

Both these problems of excessive number of accesses to the mass memory and of increased load on the processor may be alleviated to a great extent if the meta-information is separated from the database and manipulated by a functionally specialized processor. In the proposed DBC such a separation leads to the architectures of the structure memory (SM) and of the mass memory (MM). The SM (whose design details appear in [8]) contains information about the database; this information is subject to frequent changes even though the contents of the database do not change. Thus the information in the structure memory may be said to be update-variant. Examples of such information are security specifications which are different for different users, and can change with time, although the database files on which these security specifications apply may not have undergone any change.

On the other hand, consider the information contained in the mass memory. In the proposed DBC, the MM contains database files which are composed of records. Unlike records in contemporary systems, these records do not carry pointers or any other type of meta-information. Thus, any change in the meta-information does not affect the contents of the MM. We may, thus, regard information in the MM to be update-invariant. The separation of update-variant and update-invariant information provides us with three advantages: First, the number of accesses to the MM is limited to that required to access data; second, accesses to data (in the MM) may be performed concurrently with accesses to meta- (i.e., structural) information (in the SM); and, third, the complexity of the logic of either of the two components would be less than the complexity of a single component managing both update-variant and update-invariant information. This last advantage is especially important in hardware systems such as the proposed DBC.

In this section we propose a hardware organization of a mass memroy to support a database of size in the range of 10^9 to 10^{10} bytes. The principal operation carried out by the mass memory logic is the search operation. The concept of a partitioned content addressable memory (PCAM) which was successfully employed in the design of the structure memory (SM) and that of the structure memory information processor (SMIP) is once again employed here to ensure acceptable performance. The MM is designed to reduce, substantially, the effective access time to a partition. Furthermore, each partition is searched by a group of processing elements, thereby reducing the search time of a partition. These two features together with matching performance of other components will enable the DBC to be used in on-line environments.

3.1 The Design Philosophy

In the design of the SM, we used a transformation on data (keywords) to identify the sectors (or modules) of one or more partitions to which the search operation can be confined. Since the principal MM operation is the search operation, it is intuitively tempting to use a transformation of some sort to limit the search space. But such an approach, on closer examination, becomes impractical. We advance an important reason for this. First, we realize that rotating moving head magnetic recording devices are still the only cost-effective technology for large databases [9]. Given this, we are immediately confronted with the rather disparate access times and transfer rates of the device on one hand, and the large amount of sequential data available per access on the other hand. If we are to extract optimum performance from such a device, it is imperative that we place data in a way that allows information searched per access to be as close to the maximum as possible. An arbitrary transformation of data to determine its position in the MM will not, in general, result in optimum placement. However, a placement strategy based on the predicted or known data usage patterns should work better. Such a placement technique (called clustering) was introduced in Section 2 to determine the partition in which a data unit (i.e., record) must be placed. The MM design, thus, does not concern itself with where to access for a piece of data, but with how best to minimize the inherently large access times, and with how to locate (and retrieve) a piece of data (from) within the partition (i.e., MAU) defined by a single access.

To address the first of the design problems, we have proposed overlapping of access and transfer operations. It is well known that on any given device only one operation (ie., an access or a data transfer) may take place at any

given instant. However, in a system with several such devices, data transfer on one device may be overlapped with accesses on other devices. The second problem may be addressed in two different ways. In the first method, data associated with a single access may be read into a high speed random access memory and manipulated therein by a processor. The advantages of this scheme are the avilability of a large RAM which allows known software techniques to process data and that record deletions can be handled in a straightforward manner. The main drawbacks of this scheme are the rather low utilization of the RAM (typically, 3-10% of the RAM will contain information satisfying the query); and that data cannot be processed on-the-fly, since the transfer rate can be very high (160 MBS for a device with 800K bytes/sec transfer rate and 20 tracks transferring in parallel). The second method employs a set of processing elements to process data on-the-fly. Each element is equipped with a small RAM to store retrieved information. The advantages of this scheme are high utilization of all hardware resources, and the ability to process data at the maximum rate at which data can be transferred from the MM. The main disadvantage of this scheme is its inability to reclaim immediately space occupied by deleted information. The choice of the method to be employed is dictated by factors like cost and anticipated usage of the DBC, etc. If the cost of significant underutilization of hardware can be traded for instantaneous updates, then the first method is superior to the second, If, however, the delay in space reclamation is not critical, then the second method is better. We chose the second method because we believe that, although hardware costs show a downward trend, we still need to ensure adequate utilization of components and that delays in space reclamation will not be noticeable in most databases unless the MM is operating very close to its full capacity. In many contemporary systems, space is reclaimed when the load on the system is not at its peak. Such an arrangement has worked satisfactorily.

3.2 The Organization of the MM

The overall organization of the MM is shown in Figure 44. The database resides in data volumes mounted on moving-head disk drives. It is desirable to have a one-to-one correspondence between the volumes and the drives; but this is not essential, if the volumes are transferable. However, with disk technology moving towards higher bit densities, mechanical tolerances will not allow frequent interchange of volumes (disk packs) between disk drives [10]. Our design is independent of the above consideration. A volume is composed of 200-400 cylinders. A cylinder is the smallest unit of access in the MM and has been

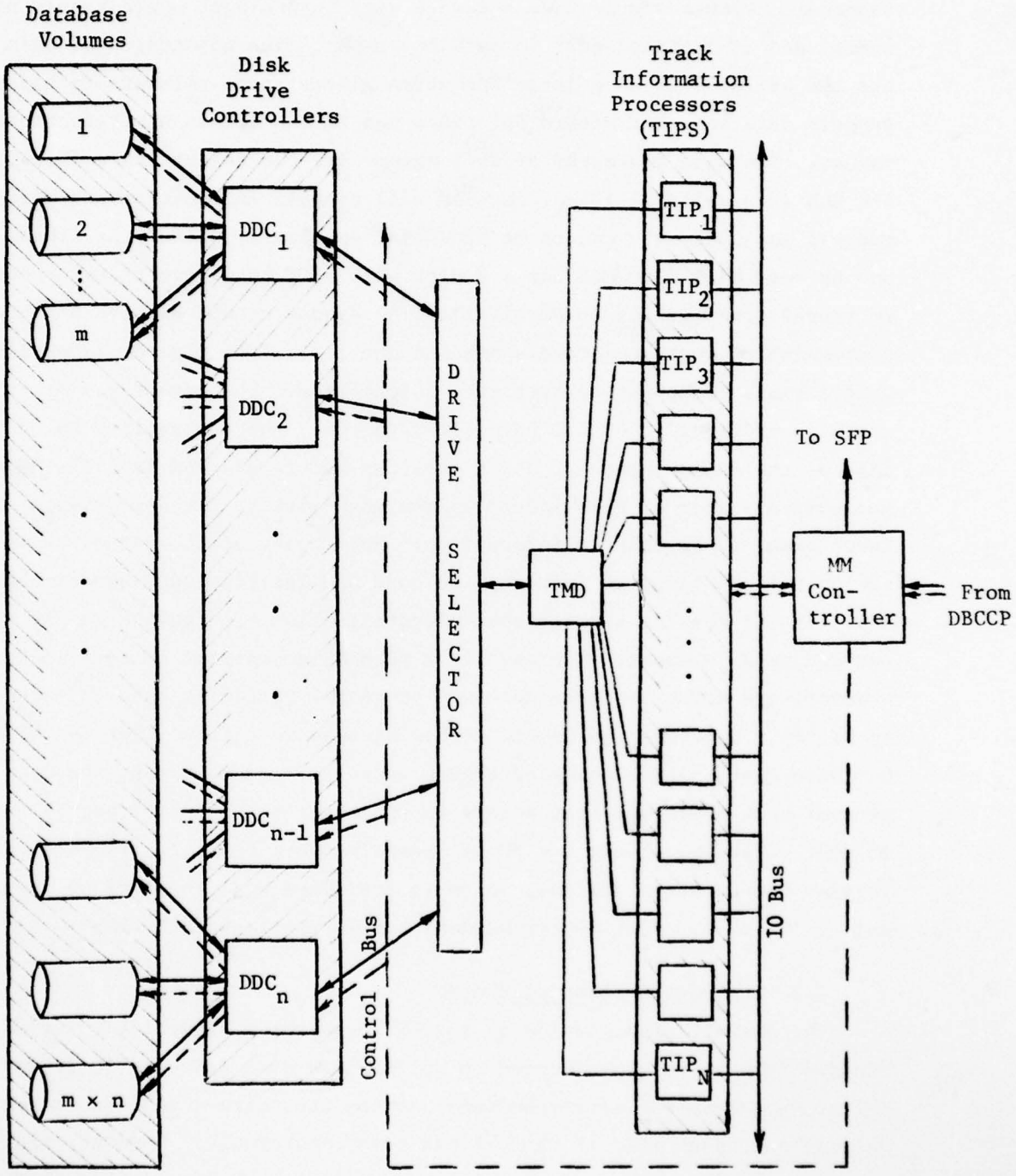


Figure 44. Organization of the Mass Memory (MM)

called the minimal access unit (MAU) [2]. Each cylinder consists of a set of tracks (usually in the range of 20-40); there is one track per disk surface. The access mechanism consists of a movable set of read/write heads, one pair per disk surface. The heads are moved in unison to access all the tracks of a cylinder. Data transfer to/from a cylinder is achieved by activating all the read/write heads concurrently. Although previous designs [11] have taken advantage of the fact that the read and write heads on a track could be positioned a short distance from each other, we do not favor such an arrangement. This is because, at high track densities (1000 tracks per inch and higher), the required mechanical tolerances may well deprive the disk technology of much of the cost-effectiveness brought about by the higher densities. In our design, therefore, we assume a combined read/write mechanism.

Each MAU in the system is uniquely identified by a number, known as its MAU address. A disk volume contains a set of consecutively addressed MAUs. The set of disk drives is partitioned into groups of 8-16 drives for access and control pruposes. Each such group is controlled by a disk drive controller (DDC). This partitioning of disk drives for access and control purposes is not to be confused with the PCAM partitioning of the data which is used for enhancing performance. The DDCs are controlled by the mass memory controller (MMC). Data that are retrieved from the disk volumes are routed to a set of track information processors (TIPs) by a drive selector and a track multiplexer/demultiplexer (TMD). The drive selector is controlled by the MMC. The TIPs can request the services of a bus called the IOBUS for transferring database information to the MMC. The IOBUS is also used by the MMC to send control information and data to the TIPs.

The MM operates in two basic modes -- the normal mode and the compaction mode. In the normal mode, orders sent by the DBCCP are decoded by the MMC and queued according to the MAUs referred to by the orders. The MMC then requests a DDC to seek the cylinder corresponding to a MAU for which a queue of requests exist. When the MAU is thus accessed, the MMC sends the orders one at a time to the TIPs. While the TIPs are busy executing the orders, the MMC can request the DDCs to position the read/write mechanisms to other MAUs for which there are non-empty queues. Thus the access time of a MAU is at least partly overlapped by useful work performed by the TIPs. The extent of overlap is determined by such factors as the average number of orders waiting to be executed for a MAU and the number of different MAUs for which there are non-empty order queues. The information retrieved by TIPs from the database is sent to the SFP for security clearance. Records which are identified by a delete order under the

normal mode are tagged by the TIPs for later removal during the compaction mode. When the DBCCP orders the MM to reclaim the space occupied by the records with deletion tags, the MM enters the compaction mode. During the compaction mode, MAUs in which tagged records exist are accessed, and data in each of the tracks is read into the MMC by the TIPs. The MMC then writes back those records which are not tagged. There are two reasons for handling deletions in this manner. First, if reclamation of space occupied by deleted records were to be attempted in the normal mode, one of two undesirable things will occur: either, we will have to provide a track-size buffer with each TIP resulting in low utilization of the buffer during retrieval, or we will have to reclaim space in segments of the track, each segment size being equal to the size of a TIP buffer. In the latter case, the number of revolutions required to sweep the entire track for reclamation will be a multiple of the ratio of the track size to the TIP buffer size. During the normal mode of operation, a single delete operation could hold up retrievals for several revolutions. This is undesirable. Second, we may expect during the course of a 24-hour day, periods of light load. Such periods usually result in low utilization of system resources. By operating the MM in the compaction mode during these intervals of light load, we may be able to achievea more equitable distribution of load on the DBC.

3.3 The Mass Memory Controller (MMC)

The mass memory controller (MMC) is organized into two subcomponents -- the interface processor (IP) and the mass memory monitor (MMM). The IP is responsible for interfacing with DBCCP, maintaining the database object descriptor table memory (DODTM), maintaining MM orders in the mass memory order queues (MMOQ), and switching from normal to compaction mode. The MMM is responsible for scheduling orders to be executed with the help of the MMOQ, issuing orders to the DDCs to position read/write heads, initiating TIPs to execute the orders on the contents of a MAU and keeping track of space availability in the MAUs.

3.3.1 Interface Processor (IP)

A. The Database Object Descriptor Table Memory(DODTM) - This table memory contains database objects which are used as arguments of the orders issued by the DBCCP. Each object is identified by a unique identification tag assigned to it by the DBCCP. A database object in this table could either be a query, a record or a pointer. The formats of these three objects are shown in Figure 45. The format of a keyword as it appears in a query or a record is also shown.

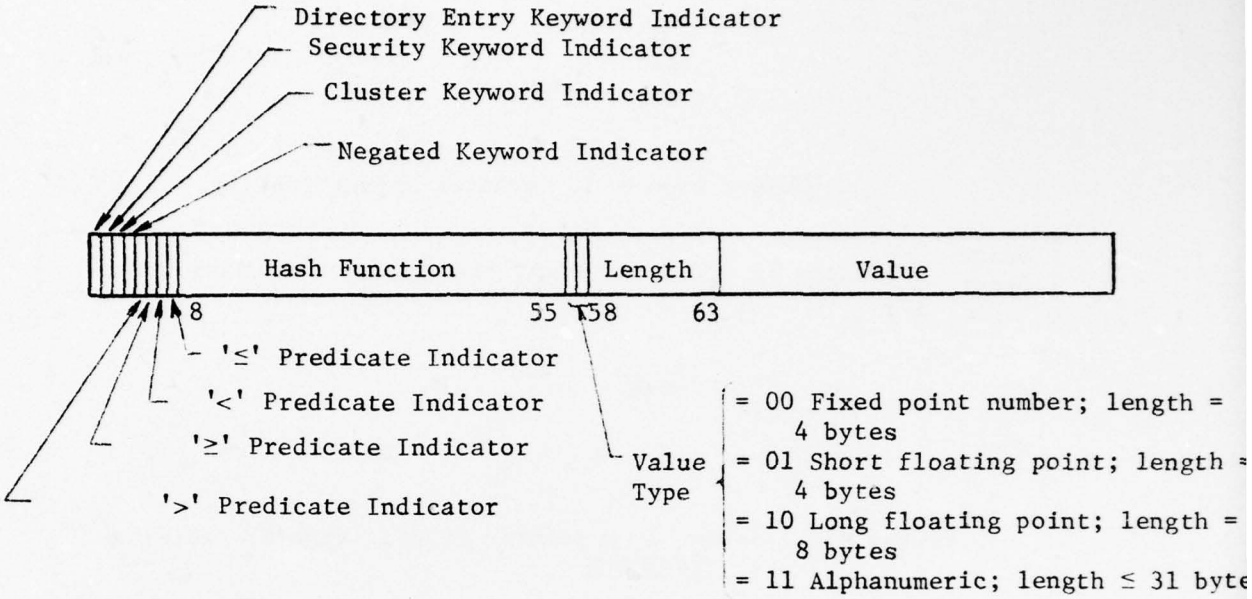


Figure 45a. Format of a Keyword Predicate in a Query or Record Sent by DBCCP as an Argument of MM Order

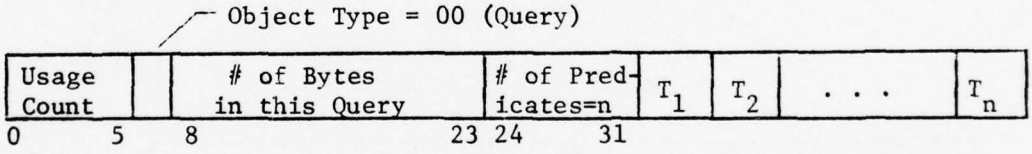


Figure 45b. Format of a Query Conjunct Residing in the DODTM

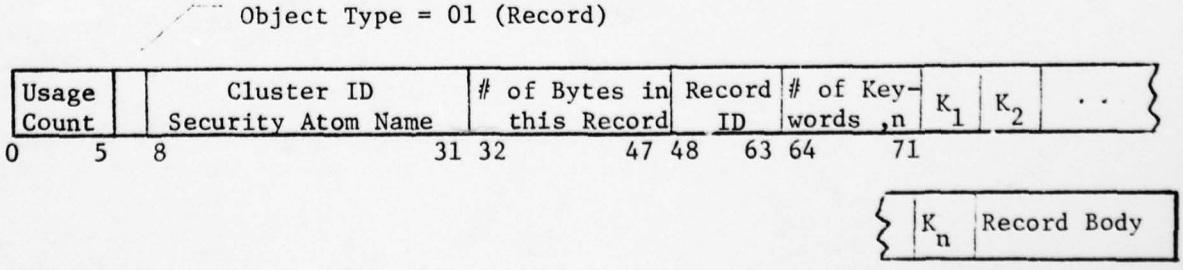
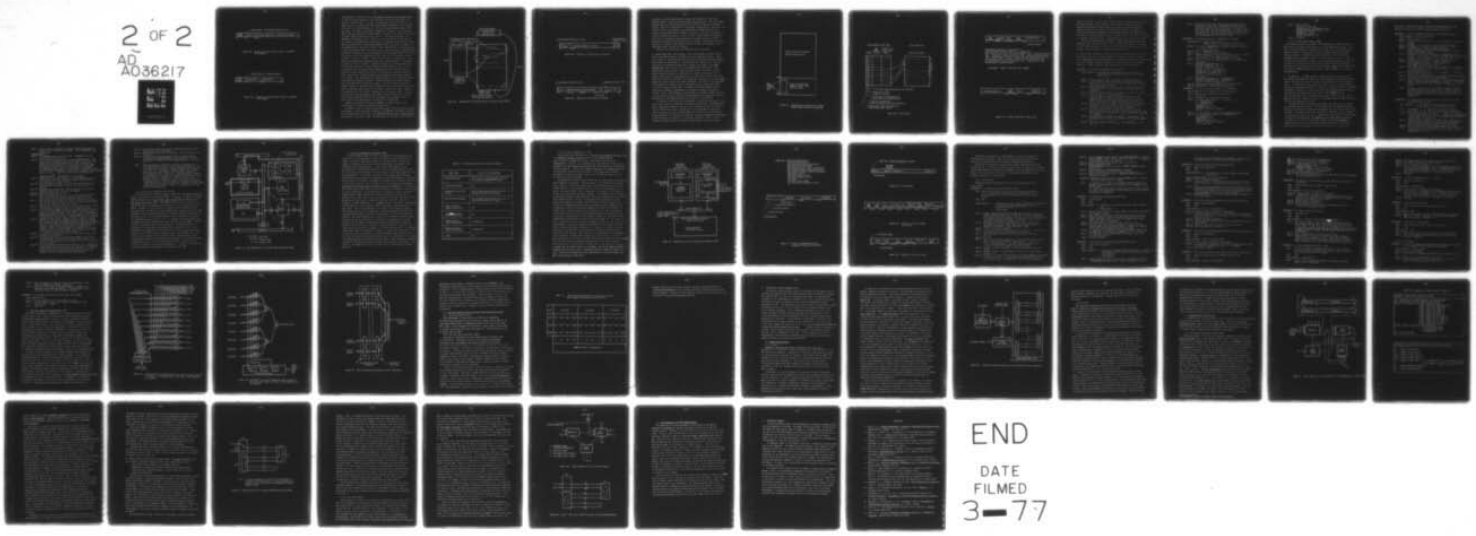


Figure 45c. Format of a Record Residing in the DODTM

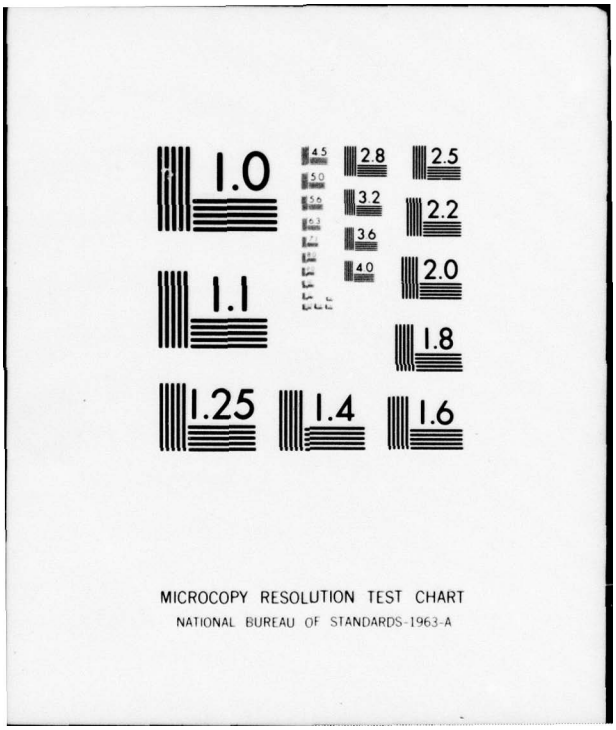
AD-A036 217 OHIO STATE UNIV COLUMBUS COMPUTER AND INFORMATION SC--ETC F/G 9/2
THE ARCHITECTURE OF A DATABASE COMPUTER. PART III. THE DESIGN O--ETC(U)
DEC 76 D K HSIAO, K KANNAN N00014-75-C-0573
UNCLASSIFIED OSU-CISRC-TR-76-3 NL

2 OF 2
AD
A036217



END

DATE
FILMED
3-77


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

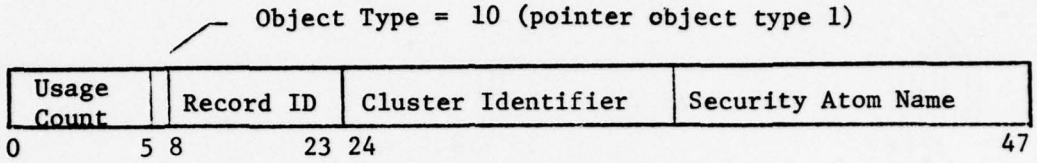


Figure 45d. Format of a pointer object (type 1) residing in the DODTM

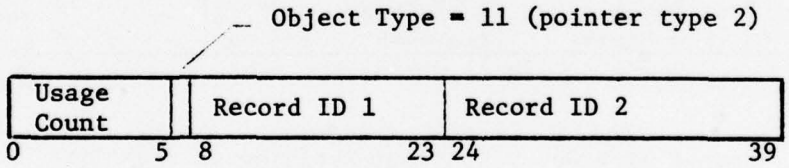


Figure 45e. Format of a pointer object (type 2) residing in the DODTM

The keywords in a query or record are assumed to be sorted in ascending order of their attribute identifier. This sorting is done by the PES before the query or record is sent to the DBC. Since database objects are placed in the table only to be accessed later when the MM order is scheduled to be executed, there must be a rapid mechanism to locate and retrieve database objects from the table. The table is, therefore, organized in two parts -- an associative memory (AM) and random access memory (RAM). An entry in the AM has two fields -- an object identification tag and a pointer to a location in the RAM. The RAM holds the database objects pointed to by the AM. The AM can be searched on the basis of database object identifiers; the response is the pointer to the RAM location where the corresponding database object is stored. In Figure 46, the organization of the table is shown. Since each MM order is associated with at most one database object and since we do not expect more than a hundred orders waiting to be executed, we can correspondingly set the maximum number of entries in the AM to be in the neighborhood of 100. Each entry in the RAM is either a query, a record or a pointer. Recall from [2] that we estimated that in the worst case, queries will seldom have more than 15-25 predicates. The same may be said to be true of records. According to Figure 45a, a keyword may occupy not more than 39 bytes (= 31 bytes value + 8 bytes overhead). Thus a query in the worst case will seldom exceed 1000 bytes in size. Records could, of course, be larger than this size, since the record body could be arbitrarily long. The size of a pointer database object (Figures 45d and 45e) is either 5 or 6 bytes. When the database has been established, (i.e., when most of the files have been created), we expect a large percentage of retrieval requests and a low percentage of insertion requests. Since only insertion requests require records as a database object argument, we may conclude that the average size of a database object would be very close to 1000 bytes. Thus, a RAM of size 100K bytes (= 100 x 1000) is appropriate for storing the DODT. The size of the AM can now be determined. Each entry occupies 4 bytes (i.e., 14 bits for the object identifier and 18 bits for pointer into the RAM). Thus the size of AM is in the neighborhood of 400 bytes (see Figure 46).

Memory in the RAM is allocated and freed in a manner similar to the scheme used for the CATM as described in Section 2.3.3.

The DODT is used in an entirely different manner during the compaction mode of operation. During this mode of operation, the MMM uses the DODT to identify those keywords whose transformed values have been deleted from a cluster-security atom partition existing in a MAU. The RAM portion of the DODT is divided into

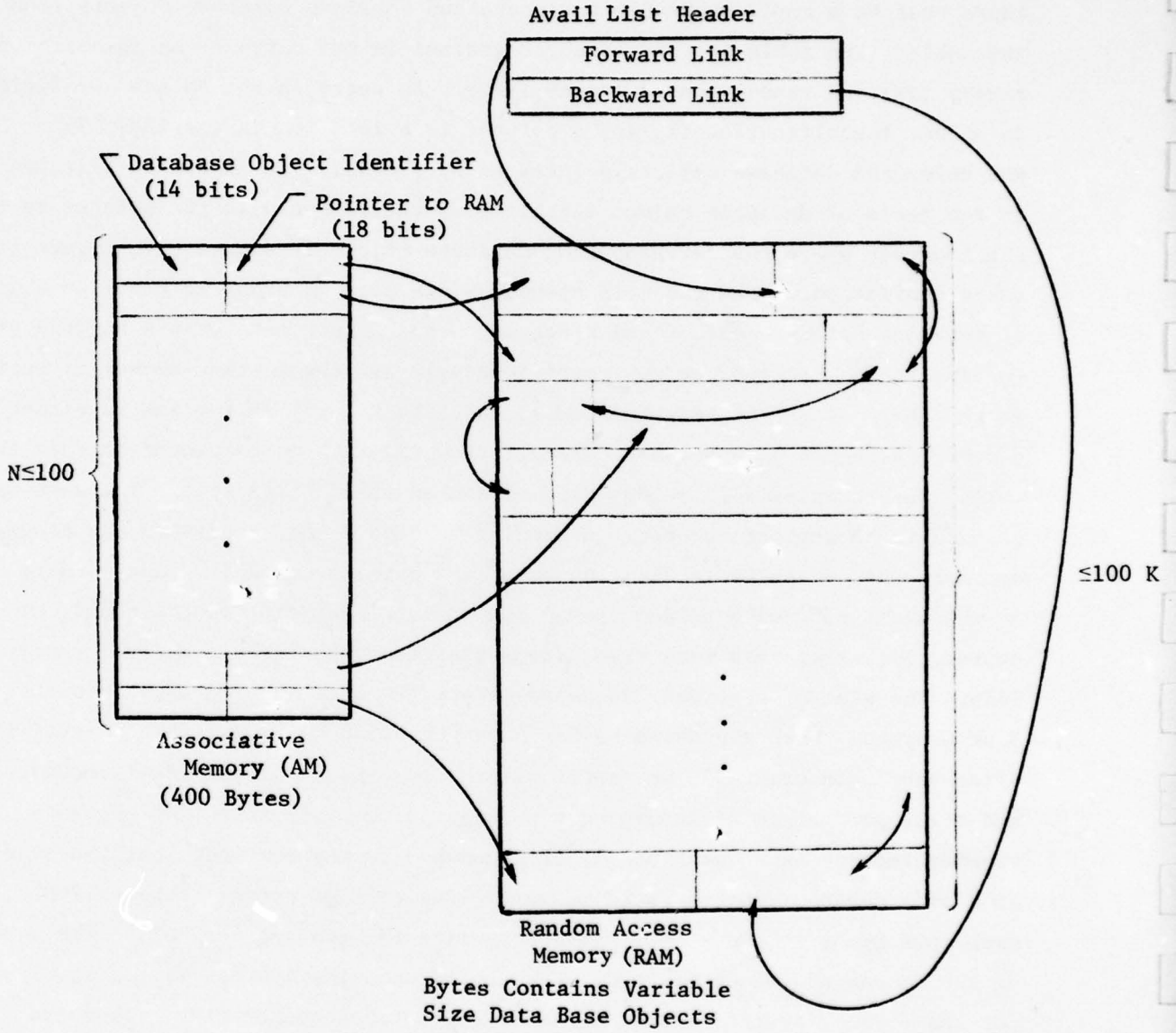


Figure 46a. Organization of Database Object Descriptor Table (DOTB)

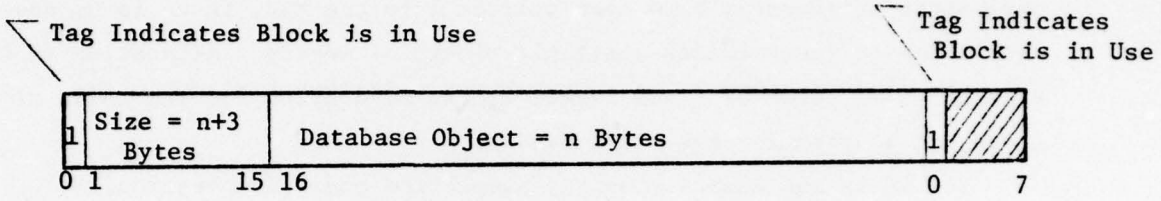


Figure 46b. Details of a Block in Use in the RAM

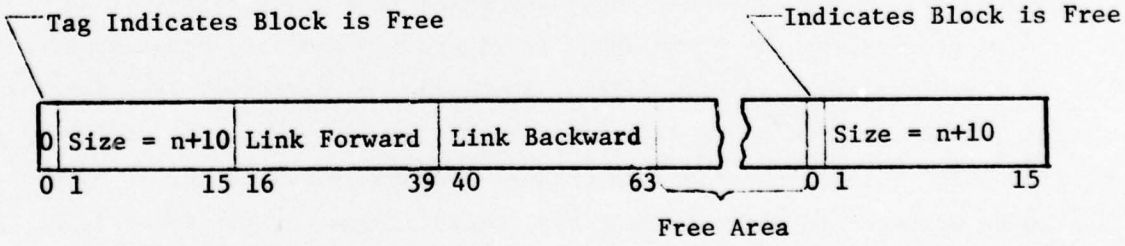


Figure 46c. Details of a Free Block in the RAM

two parts: record storage and list storage (see Figure 47). The record storage contains records from the MAU which are not deleted. The list storage contains lists of transformed keyword values which were present in deleted records. Each list corresponds to a particular cluster-security atom pair. The records in the record storage occupy about 80% of the RAM while the list storage occupies about 20% of the RAM. Allocation in the record storage is in terms of variable size blocks. Since the entire record storage is released only after the records have been written into the MAU, there is no need to keep track of intermediate available blocks of memory. Allocation in the list storage is in terms of fixed length blocks of 8 bytes for the nodes of the lists and 6 bytes for each list header.

The AM is not used during the compaction mode of operation.

B. Order Queues (OQ) - Order queues, as the name implies, are used to keep track of MM orders (sent by the DBCCP) which are awaiting execution. There is one queue for every MAU for which one or more orders are awaiting execution. Order queues are maintained on a first-in-first-out (FIFO) basis. Two data structures are proposed in Figure 48a to manage order queues. The queue headers table (QHT) is used to carry information about the queues. More specifically, each entry in the QHT has three fields: The first field has status information about the availability of a MAU for processing. The second field contains the MAU address and the third field points to the first order to be executed on the MAU. The second data structure is called the order table (OT), which contains the orders themselves. The format of an order when it is received by the MM is shown in Figure 48b and its format when stored in the OT is shown in Figure 48c. The number of different MAUs for which orders may be pending is determined by the number of entries avilable in the QHT. This in turn depends on such factors as the number of MAUs that need to be accessed in order to keep the track information processor busy, and the distribution of MAU addresses among the cylinders of disk volumes.

In order to clarify the above observation, we shall give estimates of how each of the factors can influence the number of order queues to be maintained. First, supposing the TIPs can process, on the average, an order four times as fast as a MAU access, then it follows that it is profitable to maintain at least four queues for five different MAUs each of which is in the process of being accessed by the disk drive controllers. Second, supposing the distribution of MAU addresses is localized to a particular disk drive. Then, it is not possible to overlap the data transfer and processing by the TIPs with the

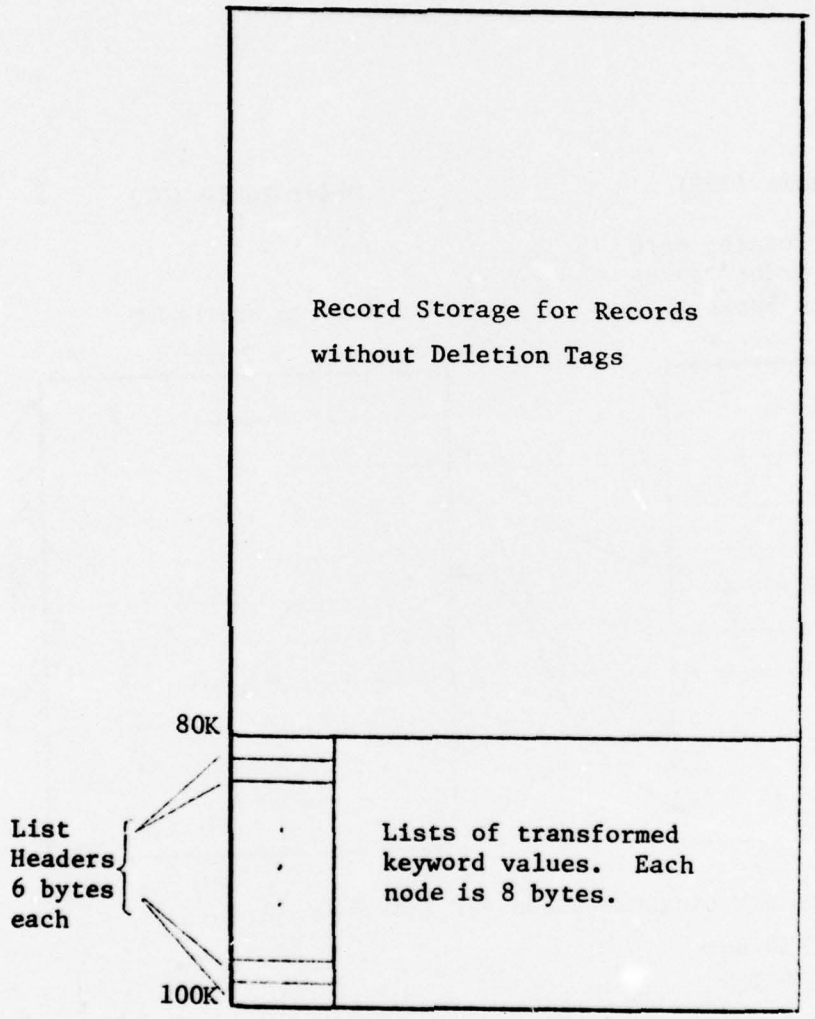


Figure 47. Organization of RAM portion of DODT during compaction mode of operation

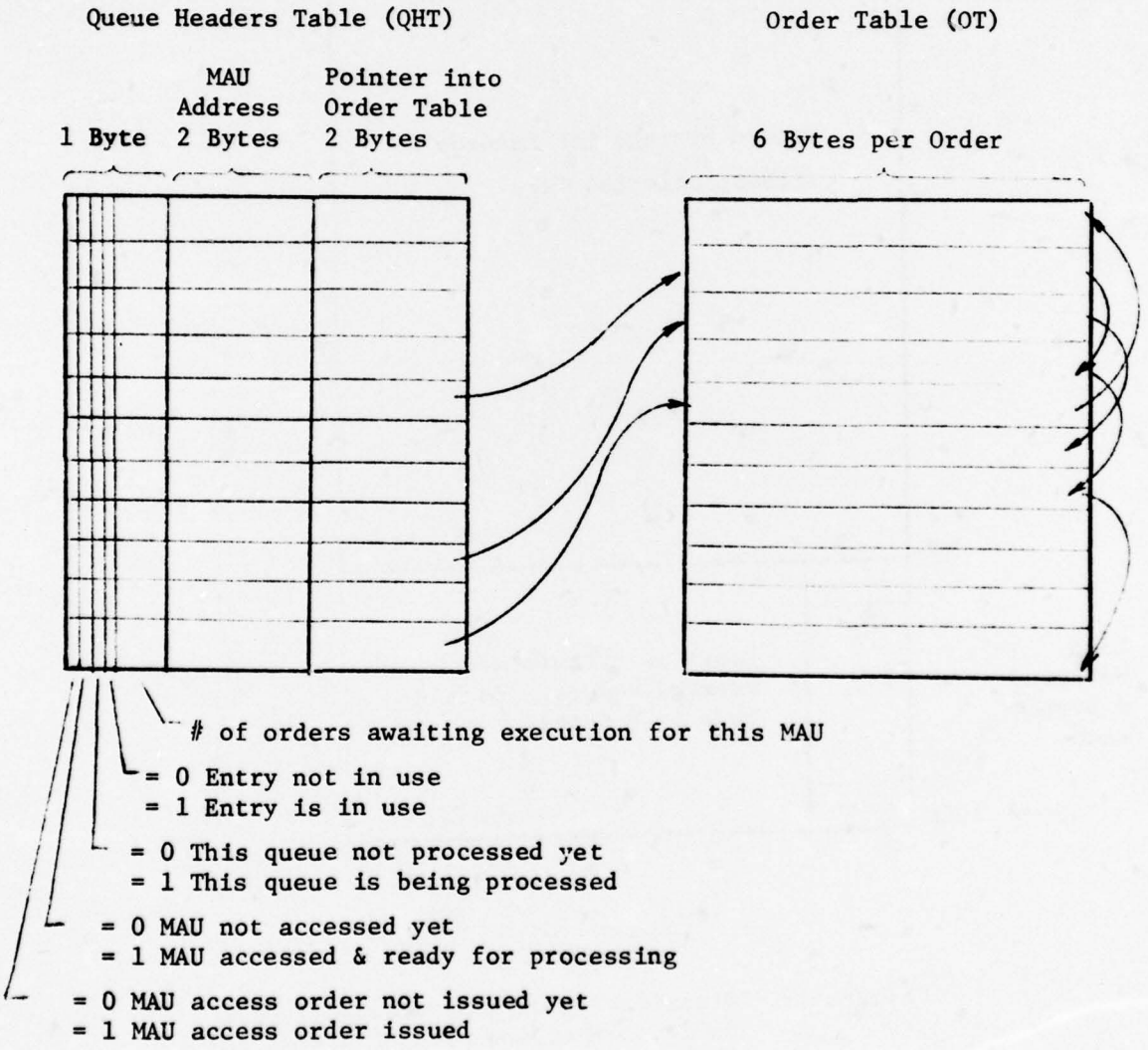
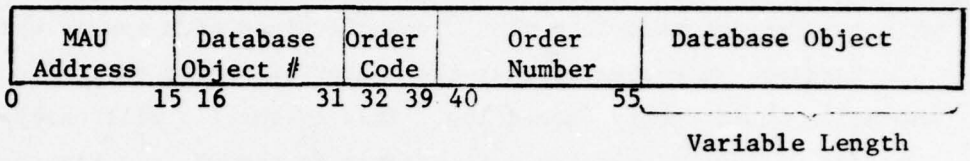


Figure 48a. Order Queues



MAU Address ranges from 0 through $2^{16} - 1$
Database Object Number ranges from 0 through $2^{16} - 1$
Order code can be: 001_8 Retrieve-by-query, 002_8 Retrieve-by-pointer, 003_8 Retrieve-by-query-with-pointer, 004_8 Retrieve-within-bounds, 005_8 Delete-by-query (type A protection), 006_8 Delete-by-query (type B protection), 007_8 Delete-by-pointer (type A protection), 010_8 Delete-by-pointer (type B protection), 011_8 Insert-record, 000_8 Reclaim-space (compaction mode).

Figure 48b. Format of MM order sent by DBCCP

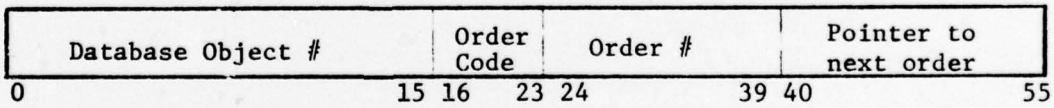


Figure 48c. Format of MM order stored in OT

access of the MAUs. In this case, it does not pay to maintain separate queues for different MAUs. Over a period of time, however, we may expect a more favorable distribution of MAUs addresses. Thus it is useful to maintain separate queues; but the extent of overlap and, therefore, the number of queues that should be maintained can be determined only by a simulation study. Each entry in the QHT occupies 5 bytes as shown in Figure 48a. Thus for a table of n entries, we need 5n bytes. Typical values of n are in the range 5-10.

Earlier, we estimated that the number of orders that might be pending execution would rarely exceed 100. This estimation will enable us to compute the size of the order table. Each entry in the OT (see Figure 48c) occupies 7 bytes. Thus, the table size need not exceed 700 bytes.

C. The IP Logic - Algorithm A below is the main algorithm executed by the IP on receipt of an order. Storage management algorithms B and C are invoked by the algorithm A to maintain the DODT. Algorithm A also detects the compaction order of the DBCCP and takes appropriate measures (in step 9) to initiate compaction of MAUs in which records with deletion tags may be present.

ALGORITHM A: To process an MM order from the DBCCP

Input Arguments: Input MM order from DBCCP in the format shown in Figure 48b and the database object used as argument of the order

Step 0: If order code is '000' (see Figure 48b) then go to step 9.
Step 1: Use the database object identifier to search the DODT (see Figure 46a). If the object is already in DODT, then increment usage count and go to step 4.
Step 2: Invoke Algorithm B to allocate space for the database object. If Algorithm B is not successful, then reject the order and terminate.
Step 3: Place the (sorted) object in the DODT in the block allocated to it in step 2. Set usage count to 1.
Step 4: Check if there is a queue for the MAU referred to in the argument order. If there is a queue, check if the MAU is being processed currently (see Figure 48a). If so, go to step 7. If there is no queue for the MAU then go to step 7.
Step 5: [Order may be added to queue] Check if there is a free entry in the OT. If not, go to step 6. Enter the order into OT and link it to the queue for the MAU. Terminate.
Step 6: [No space in OT] Reject the the order; reduce the usage count in database object in DODT. If the usage count is zero, then release space occupied by the object by invoking algorithm C. Terminate.
Step 7: [New queue to be created] Scan QHT for a vacant entry. If no vacant entry is found, go to step 6. Call the entry number 'p'.
Step 8: Place MAU address in QHT $[p]_{8-23}$. Clear QHT $[p]_{0-7}$. Go to step 5.

Step 9: [Compaction Mode] Stop accepting any more orders until compaction is completed. Extract MAU addresses from the mass memory deletion table (see Section 3.4.2) and place them in the QHT. For each such MAU place the order 'compact' with order code '000' in the order table. If QHT is full, then wait until new entries become available and then store the MAU addresses from the deletion table.

ALGORITHM B: To allocate space for a database object in DODT

Input Arguments: Size of object, say, m.
Address of AVAIL list header (see Figure 46c)

Comments: The links FORWARDLINK and BACKLINK referred to below are as shown in Figure 46c.

Step 1: Set Q ← FORWARDLINK (AVAIL).
Step 2: If Q = Λ (null), then go to step 10.
Step 3: Compare (m+3) with size of block pointed to by Q. If (m+3) is greater, then go to step 4, else go to step 5.
Step 4: [Try next block] Q ← FORWARDLINK (Q). Go to step 2.
Step 5: [Got it] If size (Q) $\geq$ (m+13) then do: P ← Q+m+3; else, go to step 7.
Step 6: FORWARDLINK (BACKLINK (Q)) ← P;
BACKLINK (FORWARDLINK (Q)) ← P;
SIZE (P) ← SIZE (Q) - (m + 3);
TAG (P) ← 0; TAG (P + SIZE (P)) ← 0;
FORWARDLINK (P) ← FORWARDLINK (Q);
BACKLINK (P) ← BACKLINK (Q);
TAG (Q) ← 1; TAG (Q + m + 2) ← 1;
Go to step 9.
Step 7: FORWARDLINK (BACKLINK (Q)) ← FORWARDLINK (Q);
BACKLINK (FORWARDLINK (Q)) ← BACKLINK (Q);
TAG (Q) ← 1; TAG (Q + SIZE (Q)) ← 1;
Step 9: Return with Q as the address of the allocated block.
Step 10: [Unsuccessful] Return with Q set to zero.

ALGORITHM C: To return a block of memory to the AVAIL list.

Input Arugments: Q, the address of the returned block

Step 1: Extract size of the block and call it m
Set Q1 ← Q.
Step 2: If TAG (Q - 1) is 1 then go to step 4.
Step 3: [Collapse lower bound] P ← Q - SIZE (Q - 1);
P1 ← FORWARDLINK (P);
P2 ← BACKLINK (P);
BACKLINK (P1) ← P2;
FORWARDLINK (P2) ← P1;
SIZE (P) ← SIZE (P) + SIZE (Q);
Q ← P;
Step 4: P ← Q + SIZE (Q). If TAG (P) = 1 then go to step 6.
Step 5: [Collapse higher bound] P1 ← FORWARDLINK (P);
P2 ← BACKLINK (P);
BACKLINK (P1) ← P2;
FORWARDLINK (P2) ← P1;
SIZE (Q) ← SIZE (Q) + SIZE (P);
P ← P + SIZE (P);

```
Step 6:  [Add to AVAIL]
         SIZE (P-1) ← SIZE (Q);
         FORWARDLINK (Q) ← FORWARDLINK (AVAIL);
         BACKLINK (FORWARDLINK (AVAIL)) ← Q;
         BACKLINK (Q) ← AVAIL;
         FORWARDLINK (AVAIL) ← Q;
         TAG (2) ← 0; TAG (P) ← 0;
Step 7:  Terminate
```

3.3.2 The Mass Memory Montior (MMM)

A. Mass Memory Deletion Table (MMDT) - The MMM maintains a deletion table to keep track of the MAUs in which there are records tagged for deletion. This table is created during the normal mode of operation and is used during the compaction mode to access the MAUs in which compaction must be performed. There is one entry in the MMDT for each of such MAUs. The first entry in the MMDT records the number of entries n that are in use currently. This is followed by the addresses of n MAUs. Each MAU address occupies two bytes, and if we do not anticipate more than N different MAUs in which deletions have been made in the time period between two compaction orders from the DBCCP, then the size of the MMDT need only be 2N bytes. Typical values of N are in the range 500-1000.

B. The MMM Logic - The MMM controls the DDCs (disk drive controllers) via the control bus (CBUS) (see Figure 44). The CBUS has an appropriate number of address lines by which each of the DDCs can be addressed to the exclusion of others. The CBUS also carries status and control lines by which the MMM can control and communicate with DDCs. The MMM also controls the TIPs via the IOBUS. The IOBUS is operated in a master-slave mode with the MMM assuming the master role and the TIPs assuming the slave roles. The IOBUS consists of bidirectional data lines over which data transfers between the TIPs and the MMM can take place, status and control lines which enables the MMM to selectively activate and interrogate the TIPs.

The MMM executes the following algorithms in the course of carrying out its functions outlined earlier. In these algorithms, all dialogues with the DDCs are carried over the CBUS and all dialogues with the TIPs are carried over the IOBUS. Algorithm A continuously monitors the QHT with a view to keeping the TIPs and the disk drives busy. Algorithm B is responsible for the detailed dialogue with the TIPs after the **algorithm** A has found a MAU that has been accessed and is ready to be processed. Among other things, **algorithm** B answers interrupts from the TIPs when they have output to be sent out of the MM or when they have finished execution of an order. Once activated by

algorithm A, algorithm B executes concurrently with algorithm A, until the list of orders for the MAU have been executed by the TIPs.

ALGORITHM A: To scan the MMOQ continuously (on a round-robin basis).

Input Arguments: QHT (see Figure 48a)

Step 1: [Initialize] $p \leftarrow 0$;
Step 2: $p \leftarrow p + 1$; if $p > N$, then $p \leftarrow 1$. [N is the number of entries in QHT].
Step 3: If $QHT[p]_3$ = '0', then go to step 2; else, go to step 4.
Step 4: If $QHT[p]_0$ = '0', then go to step 5; else, go to step 7.
Step 5: [Initiate access to MAU] MAUADDR $\leftarrow QHT[p]_{8-23}$. Decode MAUADDR into disk drive controller number d, drive number k and cylinder number c.
Step 6: Interrogate disk drive controller d, to determine if drive is free. If it is free, then issue a cylinder seek on drive k for cylinder c and set $QHT[p]_0$ to '1'. Go to step 2.
Step 7: If $QHT[p]_1$ = '0', then go to step 8; else, go to step 10.
Step 8: [Check if seek is complete] MAUADDR $\leftarrow QHT[p]_{8-23}$. Decode MAUADDR into drive controller number d, drive number k and cylinder number c.
Step 9: Interrogate drive controller d to determine if seek on drive k has been completed. If so, then set $QHT[p]_1 \leftarrow$ '1' and go to step 10 else go to step 2.
Step 10: [Initiate processing if necessary] If $QHT[p]_2$ = '0' then go to step 11; else, go to step 2.
Step 11: Interrogate if IDLE flag is on to determine if the TIPs are idle. If so, then go to step 12; else go to step 2.
Step 12: [TIPs are idle] Invoke algorithm B with the following arguments: number of MAU orders given by $QHT[p]_{4-7}$, address of first order stored in the OT for the MAU and given by $QHT[p]_{24-39}$. Go to step 2.

Note: In steps 4 through 6, we try to initiate cylinder seeks for MAUs which have not been accessed yet. In steps 7 through 9 we check on seeks already issued during a previous scan. In steps 10 through 12, we try to initiate the TIPs by invoking algorithm B.

ALGORITHM B: To initiate the execution of orders by the TIPs or accept data retrieved by the TIPs.

Input Arguments: 1. The number M of orders pending execution
2. The address of the first order in the OT

Step 1: [Initialize] $p \leftarrow 1$. Send 'reset' signal to all TIPs. [See Section 3.4 for effect of 'reset' signal on the TIPs]
Step 2: Pick up the p-th order from the OT. If order code indicates a insert-record order, go to step 6. If the order code indicates a delete order, then go to step 5, if the order code indicates a compact order, go to step 15.
Step 3: [Retrieve type] Broadcast the order (either a retrieve-by-query, retrieve-by-pointer, retrieve-by-query-with pointer or retrieve-within-bounds) to all the TIPs over the IOBUS.
Step 4: Wait for interrupt. When interrupt occurs, go to step 7.
Step 5: [Delete type] Broadcast the order (either a delete-by-query or delete-by-pointer) to all the TIPs over the IOBUS. Turn on DELETE flag. Go to step 4.

Step 6: [Insert order] Broadcast the order, find-available-space-in-track, to all the TIPs over the IOBUS. Turn on INSERTION flag. Go to step 4.

INTERRUPT ENTRY

Step 7: If INSERTION flag is on go to step 8. If DELETE flag is on go to step 9; else, go to 14.

Step 8: [Select track to place record] Turn off INSERTION flag. Read from each TIP, the amount of space available. Choose the track which has the maximum available space. Call this track i_{max}. Compute the total amount of space available on the MAU after the current record is inserted. Call this m. Also compute the maximum available space on any one track (in terms of sectors where one sector = 128 bytes). Call this n. Send the order, insert-record to $TIP_{i_{max}}$. Send the pair (m,n) to the DBCCP via the communication bus. (See Section 2) Go to step 4.

Step 9: [Check if there was any deletion] Turn off DELETION flag. If TIPs indicate that some records were tagged for deletion then go to step 10; else go to step 13.

Step 10: Store MAU address in MMDT. If order code indicates that type B protection was involved, then go to step 11; else, go to step 13.

Step 11: [Clearance by SFP required] Send the records and record IDs output by TIPs to SFP. Wait for SFP response.

Step 12: Send SFP response back to TIPs.

Step 13: Delete the order from OT. $p \leftarrow p + 1$. If $p > M$, then request TIPs to write back all deletion tags (see explanation in Section 3.4), set IDLE flag on and halt; else, go to step 2.

Step 14: [Receive retrieved records] If TIPs have records to be output, then receive them and send them to SFP. If TIPs indicate end-of-data then go to step 13; else, go to step 4.

Step 15: [Compaction] Request the TIPs to read the tagged records. [This read operation is generally done in one disk revolution.]

Step 16: As TIPs transmit tagged records over the IOBUS identify and store in an area in the DODT, the cluster numbers and security atom names in the tagged records. Also create a list of transformed keyword values for each pair of cluster number-security atom pair occuring in the tagged records. Discard the rest of the records.

Step 17: Request the TIPs to read untagged records. [Since the memory available to the MMM is smaller than the MAU capacity, the MMM will divide the TIPs into segments which are processed sequentially. Thus, if, say, 80K bytes are available to the MMM and the MAU capacity is 320 K bytes, then the TIPs are divided into 4 segments. TIPs in the same segment are requested to read their tracks concurrently, and are written into concurrently. Steps 17 through 21 are repeated for each segment.]

Step 18: As the records from the TIPs come in, store them in the record storage (one revolution).

Step 19: For each record in the record storage determine if any of the cluster-security atom pairs in the list header matches the cluster-security atom pair of the record. If so, go to step 20, else go to step 21.

Step 20: Call the matching cluster-security atom pair (c,s). Compare the transformed keyword value in the (c,s) list with the transformed keyword values in the record under consideration. If a match occurs, delete the transformed keyword value from that list. [Steps 19 and 20 take one revolution approximately)

Step 21: Write the records in the record storage back into the tracks via the TIPs. (One revolution)

Step 22: Examine the lists in the list storage. For each non-empty list do step 23.

Step 23: Transmit the transformed keyword value in the list and the corresponding triple (MAU address, cluster number, security atom name) to the DBCCP via the communication bus. Go to step 13.

Note: Since the time for executing a compaction is important, let us calculate the time on the basis of the above algorithm. The time required for reading all tagged records is one revolution [step 15]. The time required to read a segment of the TIPs is one revolution. Each segment also requires n revolutions to be processed and one to write back. Thus each segment requires n+2 revolutions to be compacted. Assuming there are m segments. Then the time for compaction of a MAU is $1 + m \times (n + 2)$. Typical value of n is between 1 and 2, and typical values of m is between 3 and 6. Thus, the number of revolutions to compact a MAU ranges from 10 to 25 revolutions. Assuming a 20 msecs disk revolution time, we obtain a figure in the range of 200 to 500 msecs for MAU compaction.

3.3.3 The Hardware Organization of the MMC

The organization of the MMC is shown in Figure 49. The internal data bus (IDB) is the main data path inside the MMC. It connects all the table memories (DODTM and OQTM) with the mass memory order argument buffer (MMOAB) and the mass memory data buffer (MMDB). The MMOAB is used to receive argument data of the MM order from the communication bus before they are transferred into the DODTM. The MMDB is used primarily as a buffer between the IOBUS on which TIPs place their retrieved data and the SFPBUS which transmits data to the SFP. The MMDB is also used during compaction as a stager between the IOBUS and the internal data bus. The interface processor (IP) is microcoded and executes the algorithms given in Section 3.3.1C. It responds to request signals from the DBCCP and controls the transfer of data from and into the MMOAB. The MM monitor is implemented with two microsequencers. Microsequencer MC-2 is responsible for executing the algorithm B given in Section 3.3.2B. It is responsible for controlling the activities of the TIPs, controlling the data transfers on the IOBUS, and data transfers to and from the MMDB. MC-2 also receives interrupt signals from the SFP and TIPs. The microsequence MC-3 is responsible primarily for scanning the order queues on a round-robin basis, initiating the MC-2, if idle and controlling the DDCs. Finally, the bus arbiter is responsible for processing requests for control of and access to the IDB and resolving contentions for the control of the IDB.

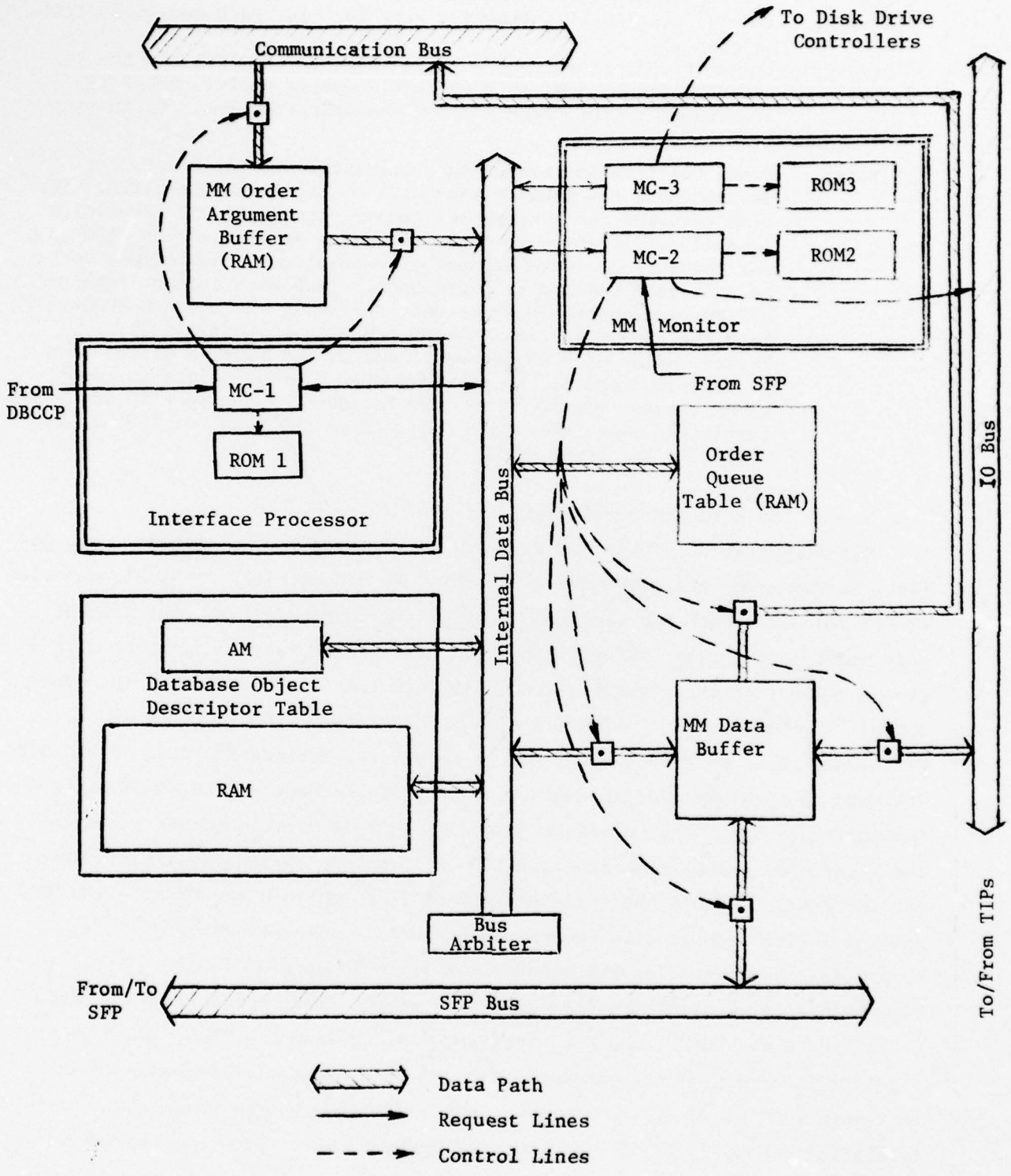


Figure 49. The Organization of the Mass Memory Controller (MMC)

3.4 The Track Information Processors (TIPs)

A track information processor (TIP) is responsible for manipulating the contents of a track belonging to a MAU. The number of TIPs is equal to the number of tracks in a MAU and is usually in the range 20-40. The TIPs are capable of searching the tracks for records satisfying a user query in one revolution of the rotating device. If the amount of information to be retrieved from a track does not exceed the size of the buffer attached to the TIP, then the retrieval operation can be performed in the same revolution as the search operation. If the buffer size is not large enough, then additional revolutions will be necessary for completing the retrieval operation. Assuming that the size of the buffer is designed to accommodate the information retrieved for most of the queries, we may conclude that, on the average, a retrieve-by-query order will require about one disk revolution for completion. A delete-by-query order always takes exactly one disk revolution for completion when the user has type A protection. As explained before, reclamation of space occupied by tagged records (compaction) does not take place during the normal mode of operation of the TIPs. When a user has the type B protection, then the records defined by the query must be cleared for security by the SFP. The SFP is designed to respond immediately to a clearance request from the MM. Nevetheless, the TIPs must wait for an unknown period of time before proceeding with the next order. Thus, a delete-by-query order usually takes longer to complete when the user has the type B protection. Retrieval-by-query-with-pointer and retrieval-within-bounds, each take on the average, close to one revolution to complete. Retrieval-by-pointer takes exactly one revolution to complete where the user has the type A protection. As before, only tagging of the record pointed to by the order is accomplished during the normal mode of operation. When the user has the type B protection, a delete-by-pointer order will take longer than one revolution to complete, for reasons mentioned above. An insert-record order takes one revolution to complete. Finally, the execution of a set of orders on a MAU is followed by one disk revoluation time during which the TIPs write back deletion information on the respective tracks (the need for this will become apparent when we discuss the TIP logic). This represents a constant overhead (of one disk revoluation) associated with each set of MAU orders. In Table IV , we summarize the times (in units of disk revolutions) for various MM orders.

Table IV. TIP Execution Times for Various MM Orders

Order Type	Time in Disk Revolutions
Retrieve-by-query	one for most queries; greater than one if retrieved information is large
Retrieve-by-pointer	one
Retrieve-by-query-with-pointer	one for most queries; greater than one if retrieved information is large
Retrieve-within-bounds	one for most queries; greater than one if retrieved information is large
Delete-by-query (Type A protection)	one
Delete-by-pointer (Type A protection)	one
Delete-by-query (Type B protection)	at least one
Delete-by-pointer (Type B protection)	at least one
Insert	one

3.4.1 The Three Components of a TIP

Each TIP has three subcomponents - the disk drive interface processor (DIP), the controller interface processor (CIP), and the buffers for the query, retrieved information (records), track header information and communications. The DIP is responsible for receiving/transmitting data as demanded by the multiplexer/demultiplexer (TMD) and carrying out the MM orders listed in Table IV. The CIP is responsible primarily for communicating with the mass memory controller over the IOBUS. Such communication involves acceptance of orders and database objects from the MMC and transfer of data retrieved by the DIP to the MMC.

The communication buffer and the buffer for the track header information are small random access memories. The query memory is a sequential access memory with a capacity to store the largest single query that may be encountered by the MM. Earlier, we had estimated this size to be in the neighborhood of 1K bytes. The record buffer is also a sequential access memory. This memory is divided into individually accessible segments. Each segment should be capable of storing the largest record that may be anticipated (in practice, records will rarely exceed 4K bytes in size). The motivation for dividing the record buffer into segments is to enable the DIP and CIP to operate concurrently, i.e., the DIP can be dumping information into one segment, which the CIP might be transmitting information from another segment to the MMC via the IOBUS. The readout rate of the query memory and the transfer rate of the record buffer should be high enough to keep in synchronization with the data transfer rate of the disk. Organization of a TIP is shown in Figure 50. The format of the communication area between CIP and DIP is shown in Figure 51. The format of a track as perceived by a TIP is shown in Figures 52a, 52b, and 52c. Each of the TIPs utilizes a bit map to remember the positions of the records which were found to satisfy the search criterion (i.e., a query or a pointer) during the execution of a delete order. Each record on the track is represented by a unique bit in the bit map. When a record is to be deleted, the corresponding bit is turned on. This bit map is stored in the first sector of the track. Before processing of a track is to begin, this bit map is read into the TIP buffer. After the last order for a MAU has been executed, the bit map is written back on the track. In processing a retrieve order, this bit map is consulted to ensure that no deleted records are retrieved. During the compaction mode of operation, the bit map is used to distinguish between tagged and untagged records. Since we don't expect more than 1000 records in a track, we need a bit map of size 128 bytes (=1024 bits).

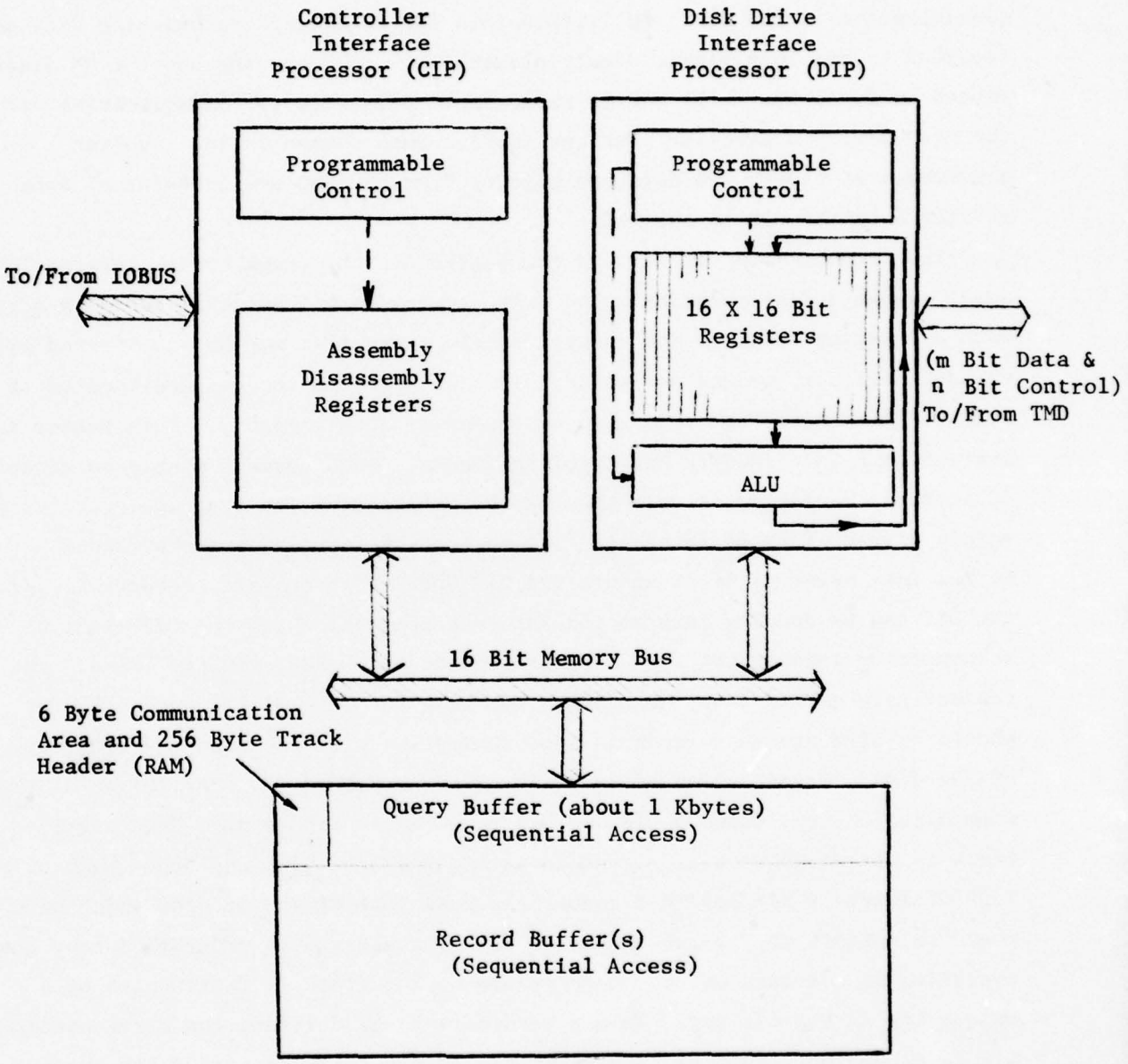


Figure 50. Organization of Track Information Processor (TIP)

ORDER CODE = 0001 Retrieve-by-query
0010 Retrieve-by-pointer
0011 Retrieve-by-query-with-pointer
0100 Retrieve-within-bounds
0101 Delete-by-query (Type A protection)
0110 Delete-by-query (Type B protection)
0111 Delete-by-pointer (Type B protection)
1000 Insert record
1001 Read Tagged Records
1010 Read Untagged records
1011 Reset
1100 Write track header
1101 Find space available on track

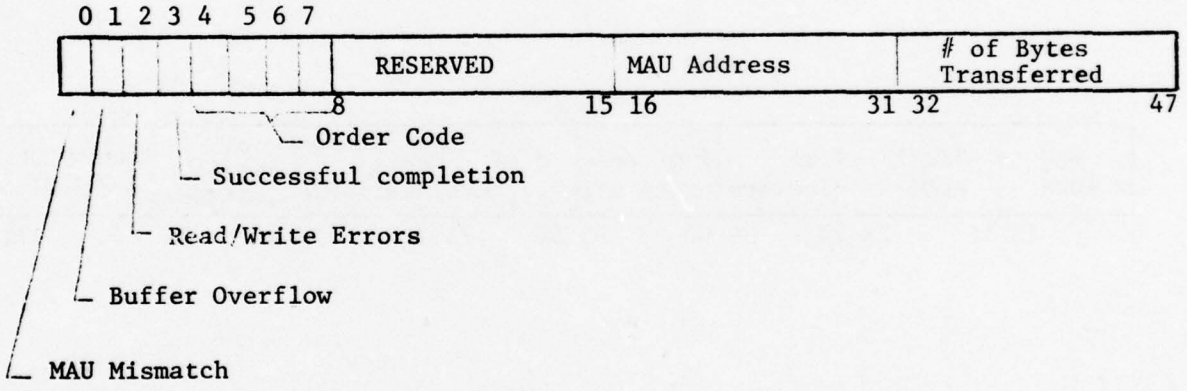


Figure 51. Format of Communication Area between CIP and DIP in TIP buffer.

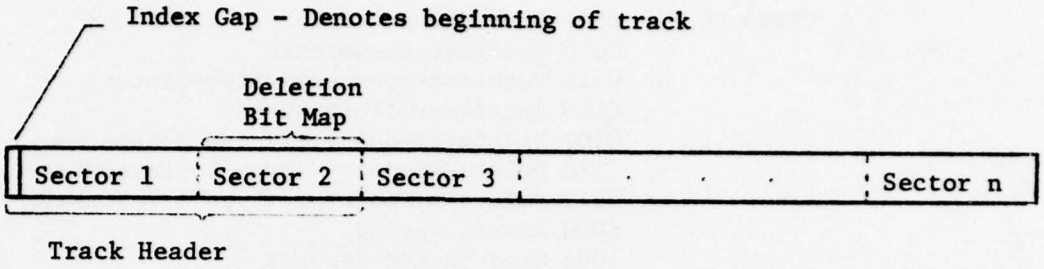


Figure 52a. Track Format

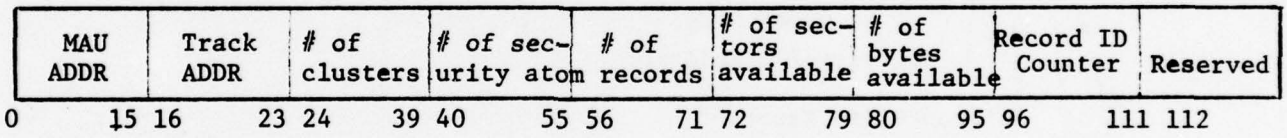


Figure 52b. Format of the first sector on a track.

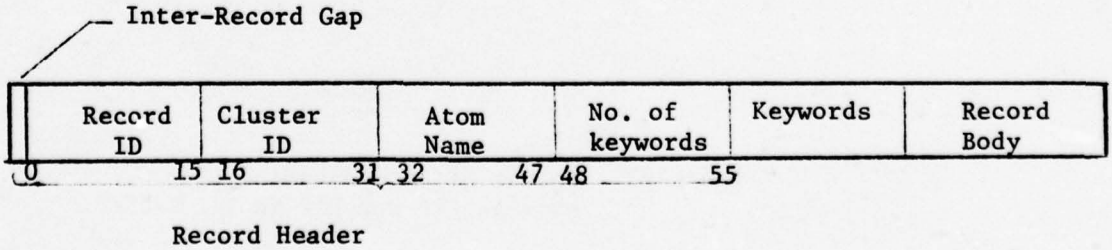


Figure 52c. Format of a record in track

Each track is divided into a fixed number of sectors for purposes of allocation (see Figure 52a). The first two sectors are used by the TIPs to store the bit map mentioned above and other housekeeping information.

We are now in a position to describe the various algorithms executed by the DIP and CIP. In these algorithms the query buffer is referred to as QBUFFER, and the record buffer is referred to as RBUFFER. When one segment of the record buffer is full (empty).the next buffer is automatically selected for data transfer, provided it is empty (full).

3.4.2 The DIP Logic

There are thirteen algorithms executed by the disk drive interface process (DIP).

ALGORITHM A: To process the retrieve-by-query order.

Time: One disk revolution for most queries; more than one revolution if a) a large amount of information is retrieved and b) the buffer segments are filled up faster than they can be emptied by CIP.

Code: 0001

Input Arguments: 1. A query in the format shown in Figure 45b. [Predicates are assumed to be in sorted order according to attribute identifier.

2. A MAU address.

Step 1: Compare argument MAU address with MAU address stored in the track header information buffer (see Figure 51). If the two addresses don't match, then reject order, Set N to 0. [N is a counter giving the number of records retrieved], and terminate; else, extract number of predicates in the query. Call it n.

Step 2: Let number of records in the track be p. [This information is available in the track header information buffer]. Set j←0. POINTER1←0; POINTER2←0. Send read signal to DDC.

Step 3: j←j+1; if j>p, then terminate.

Step 4: If the j-th bit in the deletion bit map is on (i.e., if the record is deleted) then skip j-th record and go to step 3.

Step 5: Read the number of keywords in the j-th record. Call it q; set k←1.

Step 6: Set i←1.

Step 7: Extract the attribute identifier A_i from the i-th predicate of the query. Also extract predicate type T_i ('=','≠', '≤', '<', '>', '≥') and value type V_i. If i<n, then extract A_{i+1} and see if $A_i=A_{i+1}$. If so, extract T_{i+1} and V_{i+1} and set i←i+1.

Step 8: Read the attribute identifier B of the k-th keyword in the j-th record.

Step 9: Compare A_i and B. If $A_i>B$ then go to step 14. If $A_i<B$ then go to step 15.

Step 10: [Values to be compared.] Read value from track, compare with value(s) extracted in step 7. [In the comparison, predicate types T_i and T_{i+1} and value type V_i determines the type of comparison made.]

Step 11: If the comparison is successful, then RBUFFER[POINTER1] ← keyword; update POINTER1, i←i+1. If i>n, then go to step 13. If comparison is unsuccessful, then go to step 15.
Step 12: k←k+1; if k≤q then go to step 7; else, go to step 15.
Step 13: [Record satisfies query]
RBUFFER[POINTER1] ← rest of record. POINTER2←POINTER1; n←n+1; go to step 3.
Step 14: [Wrong attribute] RBUFFER[POINTER1] ← keyword.
k←k+1; if k≤q then go to step 8; else, go to step 15.
Step 15: [Record does not satisfy query] POINTER1←POINTER2; skip j-th record; go to step 3.

Response: RBUFFER contains retrieved records; POINTER1 points to the last byte in the buffer that is occupied.

Note: 1. In step 11 or in step 13, if POINTER1 indicates overflow of RBUFFER, then processing is discontinued, and is **resumed** after one revolution.
2. In step 2, a read signal is given to the DDC to start reading the track. DDC waits until the track index gap is detected before starting data transmission.

ALGORITHM B: To process the retrieve-by-pointer order.

Time: one revolution

Code: 0010

Input Argument: 1. A pointer to a record in the format given in Figure 45d.
2. A MAU address.

Step 1: Extract record identifier, cluster number and security atom number from argument pointer.
Step 2: Send 'read track' signal to DDC [see note 2 under algorithm A].
Step 3: Compare argument MAU address with MAU address stored in track header information buffer. If the two addresses don't match, then reject order and terminate.
Step 4: Read the number of records in track. Call it p. j←1.
Step 5: From the j-th record in track, read off the record ID, its cluster number and security atom number. Compare with argument record ID, cluster number and security atom name.
Step 6: If comparison is successful (i.e., an exact match occurs), then read the rest of the record into RBUFFER, set N←1. Terminate.
Step 7: If comparison is unsuccessful skip j-th record. j←j+1. If j≤p, then go to step 5.
Step 8: Set N←0; terminate.

ALGORITHM C: To process the retrieve-by-query-with-pointer order.

Time: same as for algorithm A

Code: 0011

Input Arguments: 1. A query in the format shown in Figure 45b. [Predicates are assumed to be in sorted order according to attribute identifier.
2. A MAU address.

Step 1: Execute steps 1 through 15 except step 5 is modified as follows: "RBUFFER[POINTER1] ← MAU address, cluster identifier, security

atom name, record identifier; update POINTER1; read the number of keywords in the j-th record. Call it q; set k←1."

ALGORITHM D: To process the retrieve-within-bounds order.

Time: Same as for algorithm A

Code: 0100

Input Arguments: 1. Two pointers in the format shown in Figure 45e.
2. A MAU address.

Step 1-2: Execute steps 1 and 2 of algorithm A
Step 3: j←j+1; if j>p then terminate.
Step 4: If the j-th bit in the deletion bit map is on (i.e., if the record is deleted). Then skip the j-th record and go to step 3.
Step 5: Compare the record identifier of the j-th record with the lower bound and upper bound record identifiers in the argument pointer. If the record identifier falls in between the two bounds then go to step 6; else, skip j-th record and go to step 3.
Step 6: RBUFFER[POINTER1] ← j-th record; update POINTER1 and POINTER2. n←n+1; go to step 2.

ALGORITHM E: To process the delete-by-query order with type A protection.

Time: Same as for algorithm A

Code: 0100

Input Arguments: Same as for algorithm A

Step 1-10: Execute steps 1 through 10 of algorithm A.
Step 11: [Modified version of step 11 of algorithm A] If comparison is successful, then i←i+1; if i>n then go to step 13. If comparison fails go to step 15.
Step 12: Execute step 12 of algorithm A.
Step 13: [Record satisfies query] Set j-th bit of deletion bit map to 1. Go to step 3.
Step 14: k←k+1; if k≤q, then go to step 8, else go to step 15.
Step 15: Skip j-th record, go to step 3.

ALGORITHM F: To process the delete-by-pointer when user has type A protection.

Time: Same as for algorithm B

Code: 0101

Input Arugments: Same as for algorithm B

Step 1-5: Execute steps 1 though 5 of algorithm B
Step 6: If comparison is successful, (i.e., an exact match occurs), then set j-th bit of deletion bit map to '1'. Terminate.
Step 7: Execute step 7 and 8.

ALGORITHM G: To process the delete-by-query when user has type B protection.

Time: One disk revolution plus a variable length of time.

Code: 0110

Input Arguments: Same as for algorithm A

Step 1-2: Execute steps 1 and 2 of algorithm A
Step 3: i←j+1; if j>p, then go to step 16
Step 4-12: Execute steps 4-12 of algorithm A
Step 13: [Record satisfies query]
RBUFFER[POINTER1] ← rest of record, j.
POINTER2 ← POINTER 1; n←n+1; go to step 3.
Step 14-15: Execute steps 14-15 of algorithm A
Step 16: Wait for response from SFP.
Step 17: For each record position returned by SFP, turn on the corresponding bit in the deletion bit map.

ALGORITHM H: To process the delete-by-pointer order when the user has type B protection.

Time: One revolution plus a variable length of time.

Code: 0111

Input Arguments: Same as for algorithm B.

Step 1-5: Execute steps 1 through 5 of algorithm B
Step 6: If comparison is successful (i.e., an exact match occurs), then read the rest of the record into RBUFFER, and store the record position j into RBUFFER. Go to step 8.
Step 7: Execute step 7 of algorithm B.
Step 8: Wait for response from SFP.
Step 9: If SFP response indicates deletion is allowed, then turn on j-th bit in the deletion bit map.

ALGORITHM I: To insert a record in the track.

Time: One revolution

Code: 1000

Input Arguments: A record in the format shown in Figure 45c.

Step 1: Send "write" signal to DDC. [This signal prepares the DDC for data transfer from the TIP to the track. As in the case of the "read" signal, the DDC waits until the beginning of the track is under the read/write mechanism before accepting data for writing]. Increment record ID counter, read the counter valve and store it in the record.
Step 2: Write the record after the last record in track.
Step 3: Update number of bytes still available in track and number of complete sectors available in track. [Note that this update takes place in the TIP's local buffer area which stores the track header information. (see Figure 50 and Figure 52b)] Terminate.

ALGORITHM J: To read the tagged records on a track.

Time: If the record buffer is filled faster than it can be emptied, and if the amount of information retrieved is large, then more than one revolution will be necessary.

Code: 1001

Input Argument: A MAU address

Step 1: Verify that the argument MAU address is the same as the MAU address in the track header.

Step 2: Send "read" signal to DDC. [See note 2 under algorithm A.]
Step 3: Let number of records in track be p; N←0;
Step 4: For j = 1 through p do step 5
Step 5: If the j-th bit is turned on in the deletion bit map, then read the j-th record into RBUFFER. n←n+1. If RBUFFER becomes full, then discontinue processing for the rest of the revolution. If the j-th bit is not turned off, then skip j-th record.
Step 6: Terminate.

ALGORITHM K: To read the untagged records on a track.

Time: Same as for algorithm J

Code: 1010

Input Argument: A MAU address

Step 1-4: Execute steps 1 through 4 of algorithm J
Step 5: If the j-th bit is turned off in the deletion bit map, then read the j-th record into RBUFFER, n←n+1. If RBUFFER becomes full, then discontinue processing for the rest of the revolution. If the j-th bit is turned on, then skip the j-th record.
Step 6: Terminate.

ALGORITHM L: To process the reset order

Time: Time to read the first 2 sectors from the track.

Code: 1011

Input Argument: None

Step 1: Send "read" signal to the DOC. [See note 2 under algorithm A]
Step 2: **Read** sector 1 and 2 from the track into the track header information buffer. Terminate.

ALGORITHM M: To write back a track header.

Time: Time to write the first 2 sectors of the track.

Code: 1100

Input Argument: None

Step 1: Send 'write' signal to the DDC [see step 1 of algorithm I].
Step 2: Write track header information buffer into first two sectors of track. Terminate.

3.4.3 The CIP Logic

ALGORITHM A: To load the communication buffer with an MM order, and the query or record memory with an argument of the MM order.

Input Arguments: Data from IOBUS.

Step 1: Read order from IOBUS.
Step 2: If the order is find-space-available-on-track, then go to step 6.
Step 3: If the order is insert-record go to step 5.
Step 4: Read the argument of order and store in query memory. Place order in communication buffer. Terminate.

Step 5: Read the argument of order and store in record buffer. Place order in communication buffer. Terminate.
Step 6: Read, from track header information buffer, the number bytes available on track and the number of sectors available on the track and send them to MMM via IOBUS. Terminate.

ALGORITHM B: To transfer information from the buffer to the IOBUS.

Input Arguments: None

Step 1: For each segment of the record buffer do step 2.
Step 2: If the segment if full, then transmit the information from buffer segment to IOBUS.
Step 3: Terminate

3.5 The Track Multiplexer/Demultiplexer (TMD)

This piece of hardware is responsible for routing data between the drive selector and the TIPs. During a read operation, data from the drive selector is distributed (demultiplexed) to the set of TIPs in a round-robin fashion. During a write operation data from the TIPs is multiplexed to the drive selector also in a round-robin fashion. The time taken by the DDC to transfer a data unit to (from) a track is called the cycle time. A data unit is usually in the range 16-64 bits and is resident completely on a single track. Since all of the N tracks constituting a cylinder can be read from or written into concurrently, and since there is a one-to-one correspondence between the tracks of a MAU (cylinder) and the TIPs, it follows that the TMD must handle N data units within a cycle time. The above statement also implies that data units are always transfered to (from) a given track from (to) a corresponding TIP. The cycle time may thus be divided into N time slices; each time slice is assigned to transfering a data unit between a track and its corresponding TIP.

In order to gain an insight into the timing considerations involved, let us illustrate the above discussion with a typical example. Let us assume the following parameters: device rotating speed is 2400 rpm, track size is 16K bytes, data unit size is 32 bits and number of tracks per MAU is 20. Then time to read or write one data unit is about 6.25 μsec. This means that the time slice is about 312 nsec. If we assume no buffering in the multiplexer/demultiplexer or in the drive selector, then the propagation delay for a data unit to travel between a DDC and a TIP must be less than 306 nsec. With gate delays of 10-20 nsecs with current TTL technology, the maximum number of gates in the path of propagation of data is in the range 30-15. From Figures 53, 54, and 55, we learn that the number of gates in the path of propagation from the DDC to the TIP is 7. Amplification and inversion of signals will introduce an

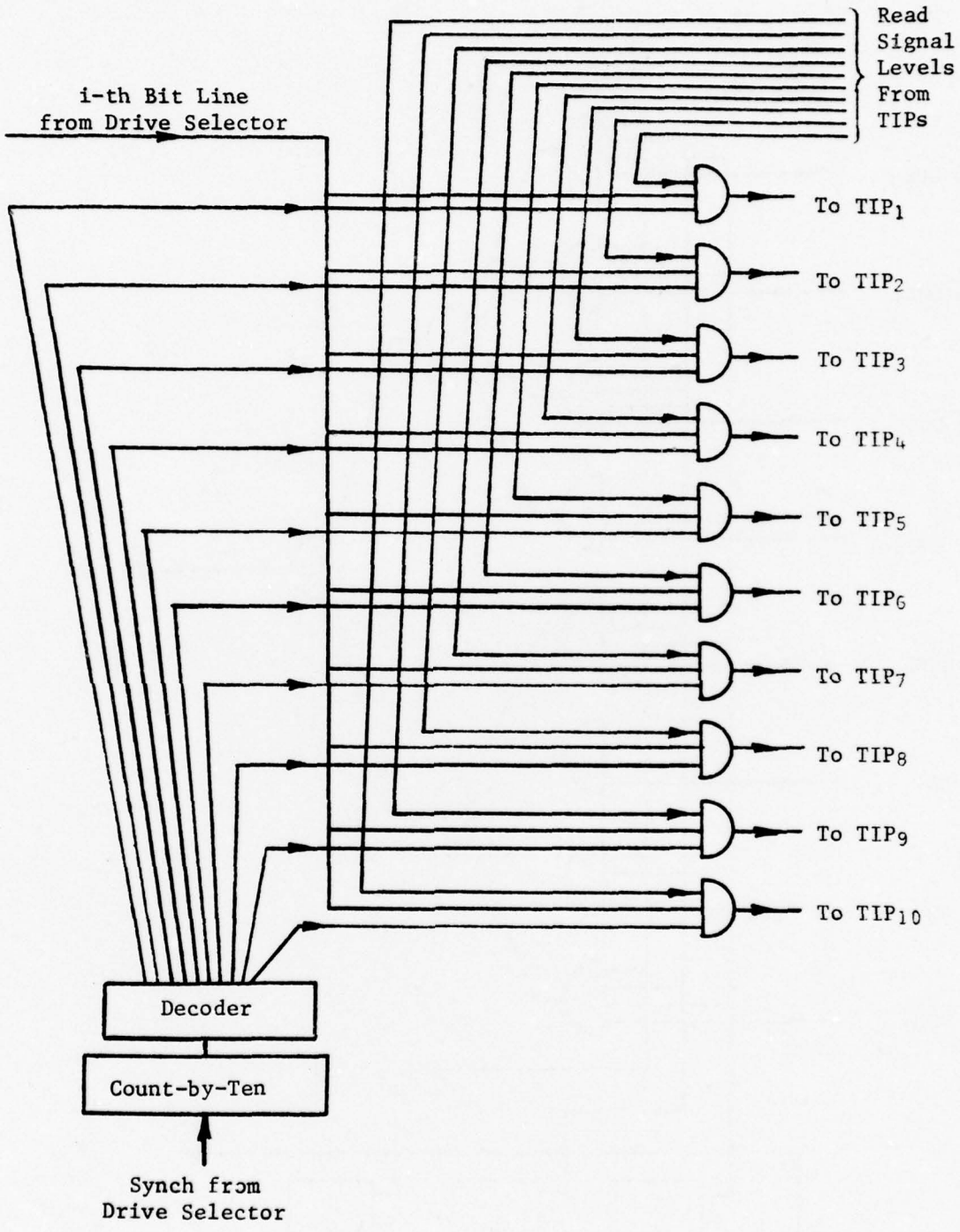


Figure 53. Logic required for demultiplexing a bit from the drive selector to 10 TIPs. If a data unit has n bits, then n such segments are required.

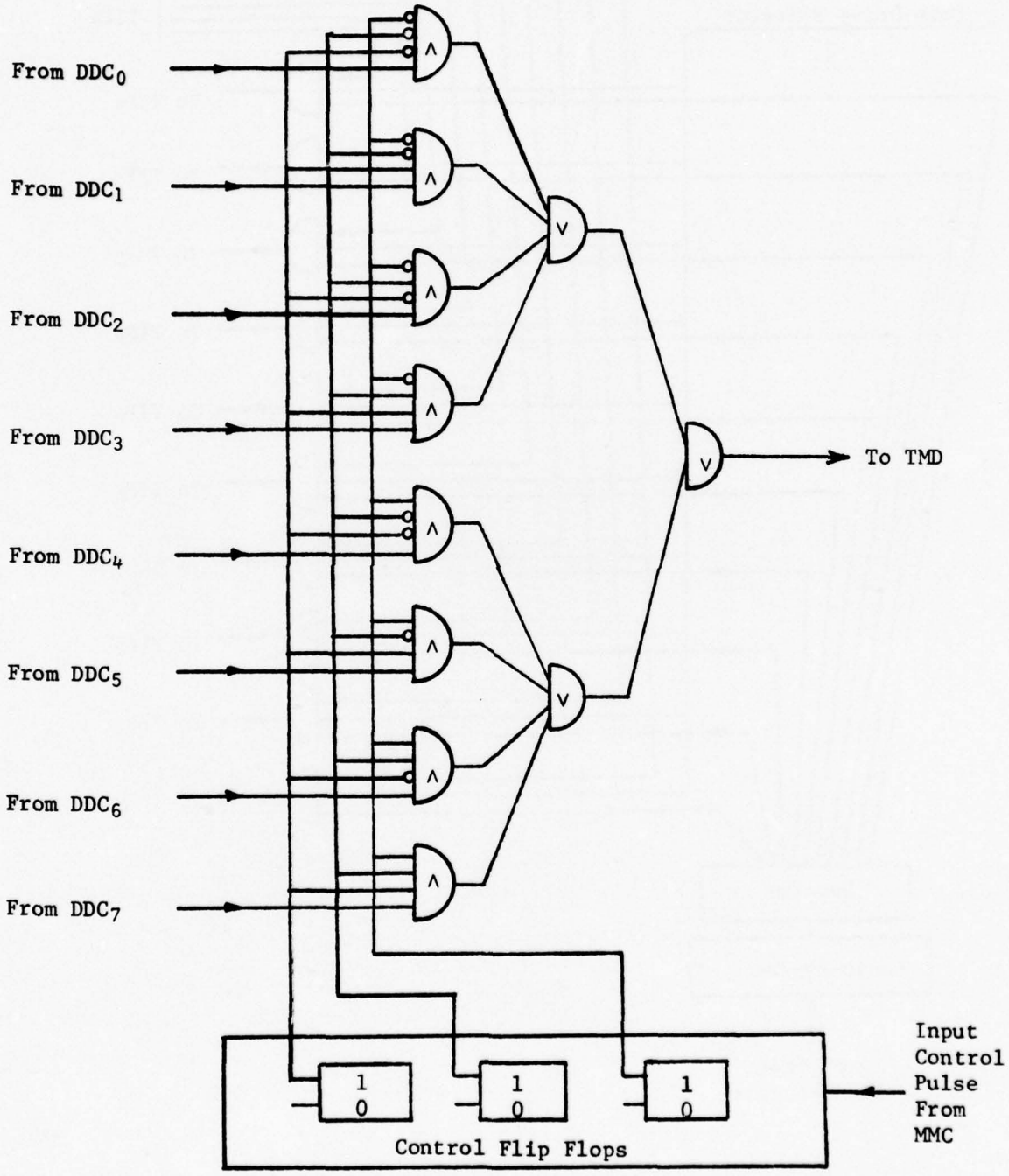


Figure 54. Propagation of one bit through the device selector. For a data unit width of n bits, n such segments will be required.

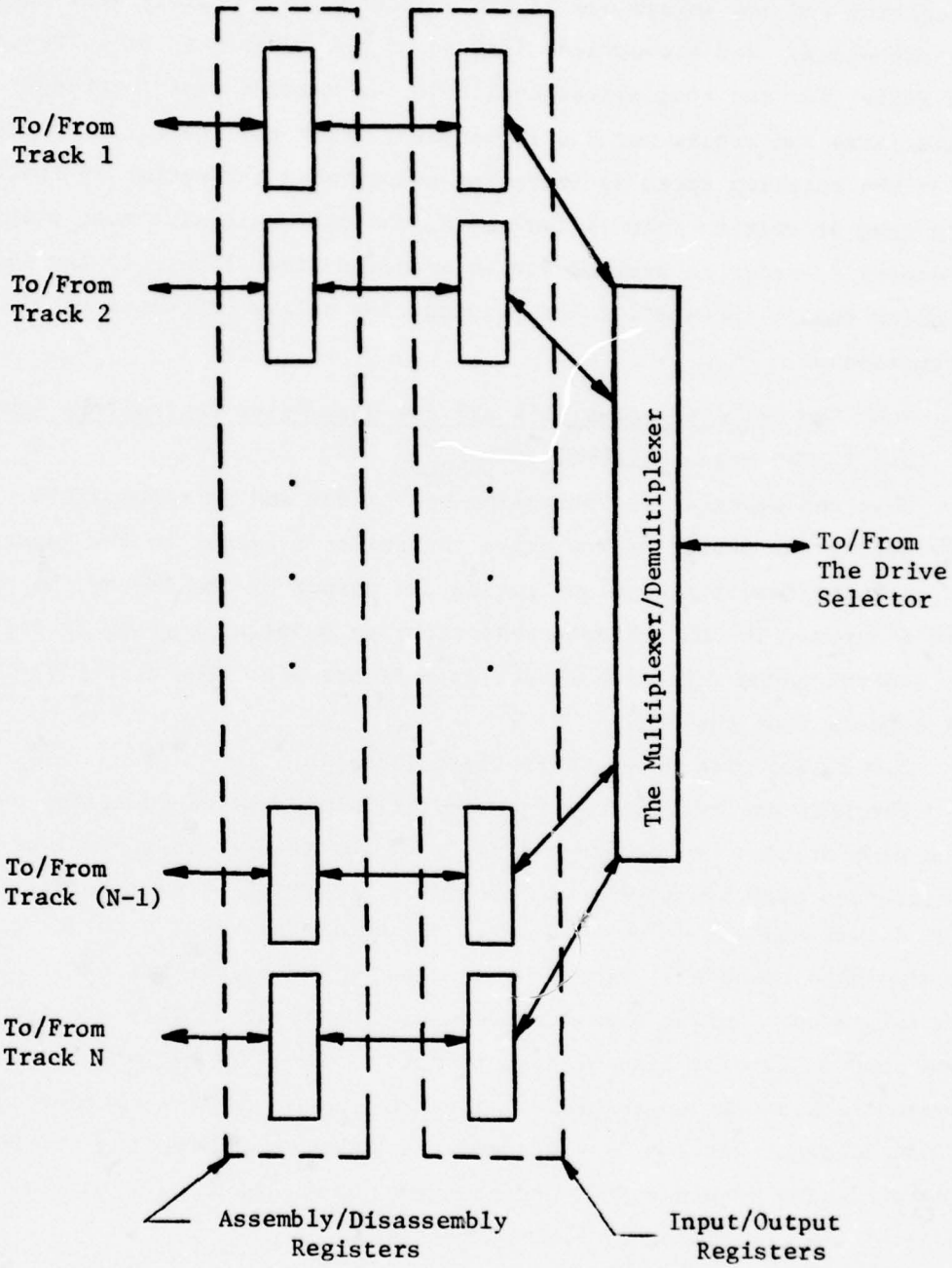


Figure 55. Flow of Information through Disk Drive Controller

additional 2-3 gate delays. Allowing 75 nsecs for data skewing at the receiving end and delays due to cable length, we are still well under the limit of 306 nsecs. For a recording density of 16K bytes per track, we have tabluated, in Table V the time slices available for various disk rotating speeds, data unit sizes and tracks per MAU (cylinder). From the table, it is clear that when the rotating speed is increased and/or when the number of tracks that are read or written into is increased, the data unit size must progressively increase in order to provide for an adequate time slice. If the time slice is smaller than a threshold value, propagation delays can cause unreliable data transmission.

3.6 The Drive Selector (DS) and the Disk Drive Controllers (DDCs)

3.6.1 The Drive Selector (DS)

This subcomponent is controlled by the MMM and is responsible for gating the proper device drive controller's output to the input of the multiplexer/demultiplexer and gating the output of the TMD to the proper DDC's input. The DS achieves this function by using a group of flip flops to control gates which route data (see Figure 54). The flip flops are set by signals from the MMM.

3.6.2 The Disk Drive Controllers (DDSs)

The DDCs are responsible for the following control functions over the disk drives: selecting a disk drive for read/write operations, initiating disk arm movements, providing buffering of data between the drives and the drive selector, and initiating error recovery procedures during data transfers. The DDCs are controlled by the MMC until a data transfer is initiated. During data transfer, signals from the TIPs are used to determine the amount and direction of data transfer. The CBUS is used by the MMM to communicate orders to the DDCs. The CBUS has an adequate number of address lines ($\log_2 m$ where m is the number of DDCs) to address any of the DDCs to the exclusion of other DDCs and status control lines which enable the MMC to monitor the activities of each of the DDCs.

Usually, a DDC controls between 4 and 16 disk drives. The DDC can initiate data transfer operation on any one of the drives and initiate arm movements on any of the drives. Since data transfer takes place on all the tracks of a cylinder, concurrently, the DDC provides for a set of assembly/disassembly of registers. There is also a set of input/output registers which can be read or written into by the drive selector. Thus, there is one pair of assembly/disassembly and input/output registers for each track of a cylinder (see Figure 55). The input/output registers serve two purposes. They serve as the interface

Table V. Time Slice in Nanoseconds as a Function of Rotating Speed, Number of Tracks and Data Unit Size

RPM	20 Tracks				30 Tracks				40 Tracks			
2400	76.3	152.6	305.2	610.4	50.8	101.6	203.2	406.4	38.1	76.3	152.6	305.2
3000	61.0	122.0	244.1	488.2	40.6	81.4	162.8	325.6	30.5	61.0	122.1	244.1
3600	50.8	101.7	293.5	496.9	33.9	67.8	135.6	271.26	25.4	50.8	101.7	203.5
	8	16	32	64	8	16	32	64	8	16	32	64
	Number of Bits in a Data Unit											

between the driver and the drive selector, and they allow data units to be assembled/disassembled while data is being read from or written into the input/output registers. The size of these registers is the size of the data unit that is chosen for safe multiplexing/demultiplexing.

4. THE SECURITY FILTER PROCESSOR (SFP)

The security filter processor (SFP) is responsible for providing security clearance for users who have the type B protection and for sorting the response data from the MM if the user has requested sorted output. As was mentioned in Section 2, the enforcement of a security policy for a user, who has the type B protection, cannot be carried out until after the information has been accessed by the MM. The information accessed by the MM is in the form of records. The security policy for a user is encoded in file sanctions contained in the user's database capability. Thus, the SFP must determine, for each of the records accessed by the MM, the file sanctions that are applicable to the record and whether the access requested by the user is granted by all of the applicable file sanctions. A file sanction is applicable to a record if the record satisfies the query contained in the file sanction. A record is said to have been cleared for security if all the file sanctions that are applicable to it grant the access requested by the user. Records, which are cleared for security, can be sorted on the basis of the values of an attribute chosen by the user, before transmission to the user. In this section, we propose an organization of the SFP which can perform the above functions in an efficient manner by employing circulating memories [12,13].

4.1 Design Considerations

4.1.1 Security

Although we mentioned above that the security policy for a user having the type B protection cannot be enforced until after the access to the MM, there is one exception. Insertion of a record can (and should) be cleared for security before the record is inserted. This implies that the DBCCP will request the SFP for a security check on all insert-record commands issued by users with the type B protection. Thus, there are two sources, namely, the MM and the DBCCP, from which requests for security enforcement may be encountered by the SFP. Requests from the MM pertain to the retrieve-and-delete commands.

Enforcement of security policies for a user with the type B protection involves comparisons of the keywords of records with the keyword predicates of the file sanctions. In order to handle the high retrieval rate of the MM in a manner that does not create a bottleneck, the SFP must be capable of performing fast comparisons. Although, conventional comparator circuits employing bipolar logic is a possible (and expensive) solution, the emergence of cheap sequential access memories with on-chip logic (CCDs) provides us with an interesting alternative.

In Figure 56, we have shown how such memories might be used to carry out several comparison operations in parallel. The operation of the scheme may be briefly described as follows: records from the MM or the DBCCP which are to be checked for security clearance are placed in a randomly accessed record memory (RM). We require a RAM to store these records, because, in general, only some of the records will be cleared for security and we need to retrieve only these records and discard the rest. It is possible that clearance of records will not be sequential but at random. A query associated with a file sanction in the database capability of the user is loaded into a sequentially accessed query memory (QM). The comparison memory (CM), which consists of a set of circulating memory-processing element pairs, is then loaded with the keywords of each of the records in the random access memory. The processing elements have two functions: searching for a particular keyword satisfying a particular keyword predicate or acting as a connector between two adjacent data paths. The first function is useful in determining if a record satisfies a query, and the second function is designed to handle the variable length of the keywords and the variable number of the keywords in a record. The keywords in the record and the keyword predicates in the query are assumed to be in sorted (ascending or descending) order of attribute identifiers. The memory controller (MC) reads a keyword predicate from the query memory, and broadcasts this information to all the processing elements which are not acting as connectors. The fact that keywords are ordered enables the processing elements to search only up to a point (in the circulating memory) where the attribute identifier is greater than the search attribute identifier supplied by the memory controller. At the end of one complete circulation time, the comparison memory would have determined which of the keyword sets satisfy the query and which of them do not. On the basis of this knowledge, the memory controller can proceed to determine if the access requested by a user is permitted by a file sanction on the records satisfying the file sanction query. Those records on which the access is denied are deleted from the random access memory. By repeating the above procedure with each of the file sanctions in the database capability of the user, the memory controller can determine the set of records on which the desired access is permitted by the database capability.

Since security enforcement is often regarded as an overhead, it is important for us to have an idea of the time taken by the SFP to perform the operations discussed above. Using CCD technology, it is possible to obtain up to 10 MBS shift rate. Assuming that in the worst case, file sanction queries are unlikely to have sizes more than 1K byte, the time required to determine whether

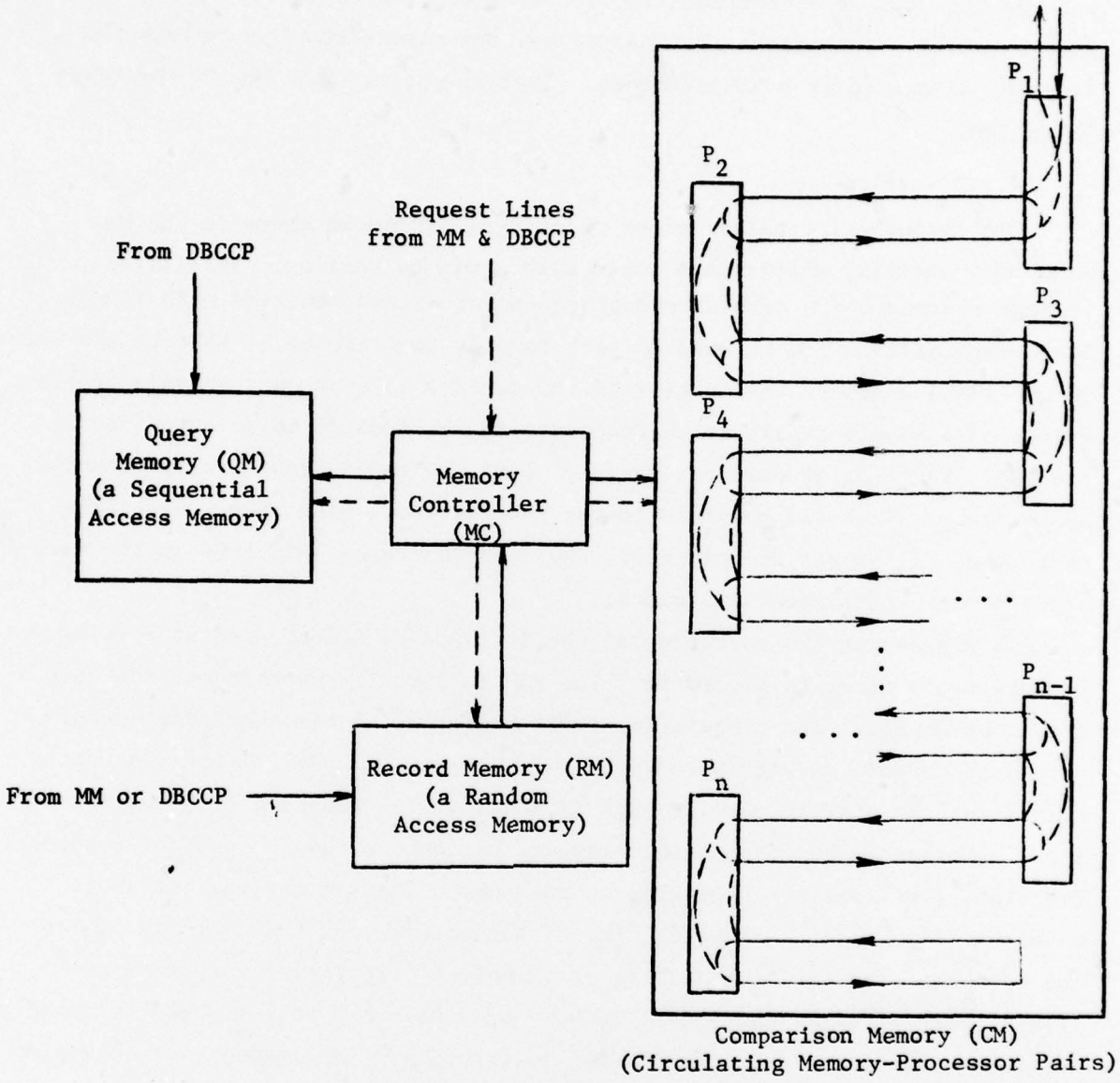


Figure 56. A Scheme to Utilize Sequential Access Memories for Fast Comparison

records satisfy a query or not, is of the order of 1 msec. Thus, if the database capability contains n file sanctions, the time taken to complete the security clearance is n milliseconds. Typical values of n lie in the range 10 to 100.

4.1.2 Sorting

The technology considerations which were discussed above in the design of a security enforcement schem also apply to sorting. Intelligent storage systems which combine cheap sequential access memories with simple logic capabilities can be used to sort records that are to be sent to the user. As mentioned earlier, the sorting of the records is done on the basis of the values of a single attribute which occurs in the records to be outputted to the PES. Thus, it is only necessary to load the intelligent memory with the appropriate values and pointers to the records containing them. After the sorting of the values is completed, the records can be outputted in the sorted order of the attribute-value pairs.

In discussing the operation of the intelligent memory used in sorting, we once again refer to Figure 56. The RAM is used for storing records which are to be sorted. The circulating shift registers in the comparison memory will each contain an attribute value and a pointer to the RAM where a record containing the attribute-value pair is stored. The memory controller can issue commands to the processing elements to compare the contents of adjacent circulating registers. Depending on the sort criterion and the attribute values, the processing elements will either interchange the contents of the two adjacent circulating registers or allow the circulating registers to retain their original contents. Such an operation may be performed repeatedly until the contents of the circulating registers are in sorted order according to the attribute-value pairs. The query memory is not used in this sequence of operation. The maximum time taken to sort n circulating registers have been shown to be $\frac{n \cdot r}{2}$ where r is the circulation time [11].

From the above discussion, one might gain the impression that the same piece of hardware may be used for both security checks and sorting of records. While such an approach is entirely feasible, it is not desirable. We advance several reasons for this. First, from the point of achieving maximum concurrency within the SFP it is desirable to have separate units for security checking and sorting. Under this scheme, records which have been cleared for security may be sent to the sorting hardware for sorting, thus freeing the security hardware to handle the next batch of records for security clearance. Second, although,

in principle, the hardware for comparison operation and sorting are similar, they are by no means identical. If we use the same hardware for both operations, we would need to design features which may be required for one or the other operations but not both. An example is the query memory which is used in security checks but not in sorting. Several other features will be discussed in the next section. Third, deletion in the MM cannot be completed (for users with type B protection) until security clearance is obtained from the SFP. This reinforces the need for concurrency discussed above. Thus, we conclude, that we should have separate units for security checks and for sorting.

4.2 Implementation Considerations

In accordance with the discussions in the last section, the SFP is implemented using two modules which can function concurrently. These are the security enforcement module (SEM) and the sort module (STM). In this section, we present details of the components of the two modules and how they interact to produce the desired result, namely security enforcement and sorting.

4.2.1 The Security Enforcement Module (SEM)

A. Processing Element - By far the most important component of the SEM is the processing element (PE) in the comparison memory (CM). In Figure 57, we have shown the five components that comprise a PE. These are comparators, the path* control, the data routing logic, the timing control and the argument registers. The comparator makes serial comparisons between the contents of a path and those of the P and Q registers. The type of comparison is specified by the four search specifier lines S1-S4. The lines S5 and S6 specify the value type of the comparison arguments. The path control is responsible for proper connections between adjacent paths. When the PE is in the search mode (control line C= o), then the path control's function is to maintain the connection between L_i and L'_i and L_{i-1} and L'_{i-1}. When the PE is in the connect mode (control line C = 1), then the path control's function is to maintain the connection between L_i and L'_{i-1}, and L_{i-1} and L'_i. The data routing logic directs data from the input data line D1 to either of the two registers in the PE or the path L_i. When the control line C_i of PE_i is low (= 0), then loading of P, Q or path L_i can be achieved by maintaining one of L1, L3, L3 high. If C_i is high (= 1) no loading can take place. The timing control unit is responsible for generating all the signals at the appropriate times for the correct functioning of the three units described above. The timing control unit takes as input the signals C_i, L_1, L_2 and L_3 generated by the memory controller. The fifth component of the PE consists of four sequential access memory elements - A, B, P and Q registers.

*A circulating register is called a path in this discussion.

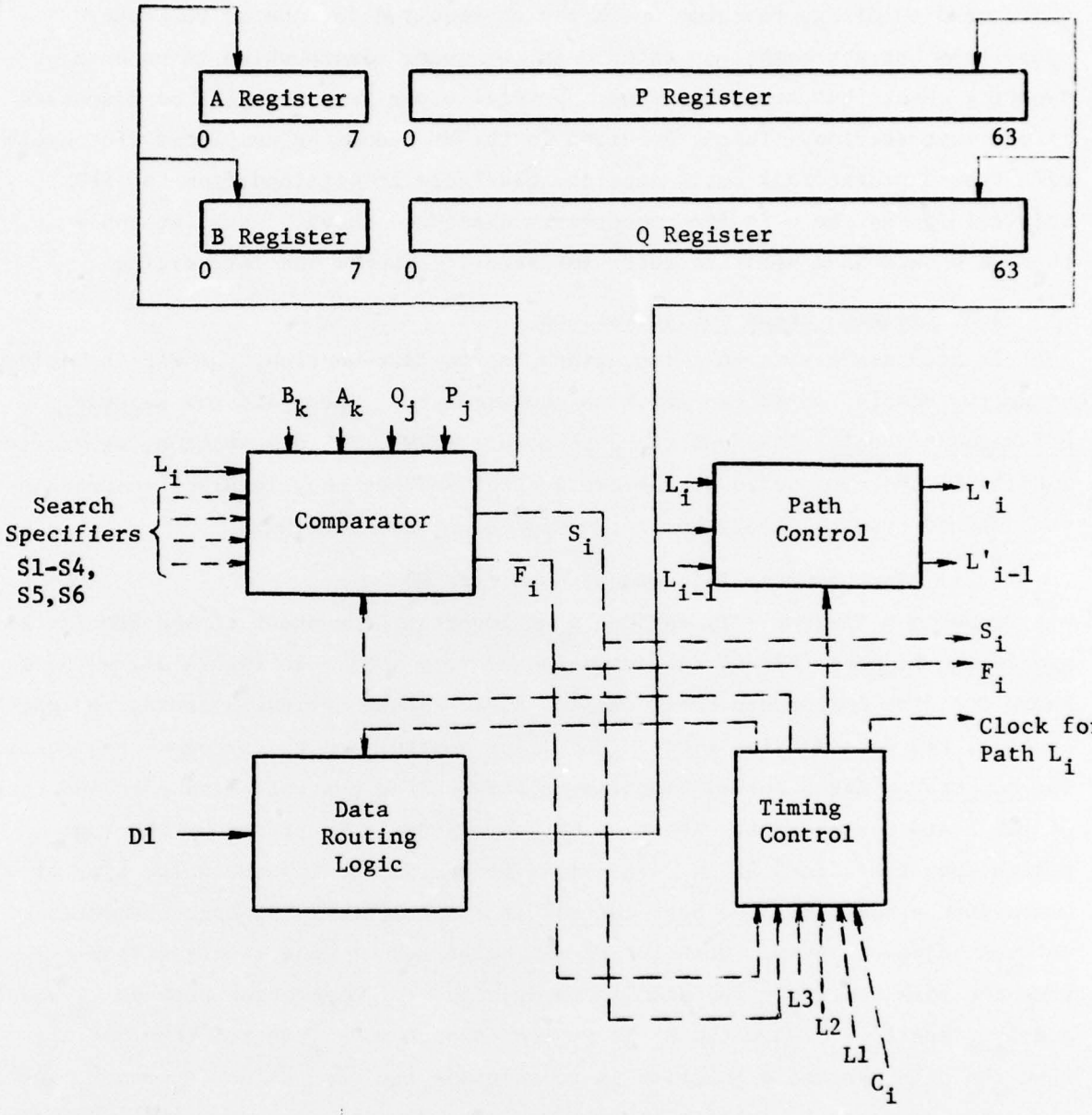


Figure 57. Block Diagram of a PE (see Table IV for explanation of control lines)

TABLE VI. Explanation of Control lines in Figure 57

Broadcast Control Lines (common to all PEs)
L1: Load path; L2: Load P-register; L3: Load Q-register D1: Serial data for P and Q registers and path L_i S1-S4: Search Specifiers = 0001 - Attribute Search = 0010 - Search for value equal to P = 0011 - Search for value $\neq$ P = 0100 - Search for value $\leq$ P = 0101 - Search for value $<$ P = 0110 - Search for value $\geq$ P = 0111 - Search for value $>$ P = 1000 - Search for Q $\leq$ value $\leq$ P = 1001 - Search for Q $<$ value $\leq$ P = 1010 - Search for Q $\leq$ value $<$ P = 1011 - Search for Q $<$ value $<$ P = 0000 - Comparator inactive S5,S6: Value type specifier = 00 fixed point = 01 floating point short = 10 floating point long = 11 Alphanumeric D1: serial data input

Individual Data and Control Lines (one of each line for each PE)
L_i - Input from data path i L'_i - Output to data path i L_{i+1} - Input from data path i+1 L_{i+1} - Output to data path i+1 C_i - Connect control line = 0 L_i connected to L'_i, L_{i+1} connected to L_{i+1} = 1 L_i connected to L'_{i+1}, L_{i+1} connected to L'_i S_i - Search successful indicator F_i - Search failure indicator

A and B registers are called difference registers, and P and Q registers are known as value registers. The value registers can be loaded from the data line D1. The difference registers are used by the comparator in **floating point comparisons.**

We are now in a position to describe how the PE is used to carry out a search operation for a set of keywords belonging to a record satisfying a query conjunct of keyword predicates. Assume that a set of keywords has been loaded into the i-th data path. (We shall describe the loading operation later in this section.) The memory controller orders PE_i to get into the search mode ($C_i = 0$). The P register is then loaded (L2 = 1) with **the attribute** identifier. The search specifiers are then set to "attribute-search" (S1-S6 = 0001) by the memory controller. The comparator compares the next 16 bits with the 16 bits in the data path L_i. If the bits match then the signal S_i is raised and path movement is halted to await the next instruction from the controller. If the data path contents are greater than those of the P-register, then the F_i signal is raised. If the data path contents are less than those of the P-register, then the data path L_i is shifted beyond the value bits following the attribute identifier. The shift count is maintained by the A register which is loaded with the length of the value field in the data path. [Recall from Figure 4 that the length of a keyword is recorded between the attribute and the corresponding value] A comparison is again attempted with the next attribute identifier in the data path. If the end-of-path is reached, without an attribute match, then the line F_i is raised. When an attribute match is obtained by PE_i, then the memory controller initiates a value comparison between the value in the keyword predicate and the value in the dat path. The lines S1-S6 are set accordingly, and the registers P and Q are loaded before the comparison begins. Fixed point comparison is straightforward. In the case of floating point comparison, the comparator has to remember the exponent difference between the comparands, since the values need not be normalized. The A register is used to store the difference between the exponents of the P-register value and the data path value, while the B-register is used to store the difference between the exponents of the Q-register value and the data path value. The mantissas are then compared. During the comparison, additional zeros are inserted in front of either the P-register bit stream or in front of the data path bit stream depending on the sign of the difference stored in A-register.

Sometime, the query conjunct has two keyword predicates belonging to the same attribute. This implies a between-the-limit type of search. In such cases,

the memory controller loads both the P and Q registers with the upper and lower bounds of the search. The PE then carries out simultaneous comparisons between the contents of the P and Q registers on one hand and the data path on the other. The signal S_i is raised by the comparator if the comparison of values is successful; the signal F_i is raised if the comparison is not successful.

Once the F signal is raised by a PE, it cannot participate in further search operations until the data path contents are replaced ($L_1 = 1$). When the memory controller has processed all the keyword predicates in the query or when all the PEs have shifted their respective data paths through a full cycle, the controller reads the S-signals of all the PE elements which were in the search mode. The records whose sets of keywords were processed by PEs with the S-signal **true** satisfy the query conjunct in the query memory.

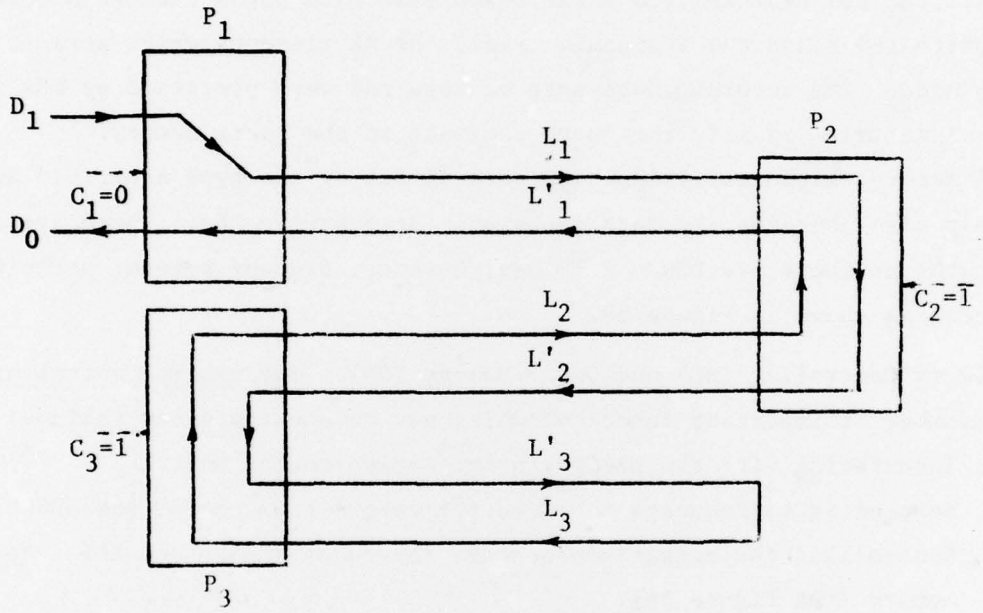
A memory chip usually contains several PEs of the type described above. The chip also contains the data paths processed by the PEs. There are as many data paths as there are PEs. A PE may, however, process several paths connected in tandem as shown in Figure 58.

B. Memory Controller (MC) and Query Memory (QM) - The memory controller (MC) has a number of important functions which may be enumerated as follows:

- Interfacing with the DBCCP via the communication bus.
- Responding to requests for security checks from the MM and DBCCP.
- Controlling the comparison memory, the query memory and the record memory (see Figure 56).

The MC needs file sanctions belonging to a user database capability in order to enforce security. These file sanctions are stored in the security information table memory (SITM) in the DBCCP. The MC is able to access the SITM by competing for control of the communication bus with the processing components of the DBCCP. The MC will accept requests for security checks from the MM and the DBCCP, with the MM enjoying a higher priority than the DBCCP. The MC controls the comparison memory by issuing the correct sequence of broadcast signals (see Table VI) and by monitoring the response signals of the comparison memory. The MC controls the query memory by overseeing the loading of file sanction queries into the query memory and by reading out keyword predicates from it as arguments to the comparison memory. The record memory is used by the MC to store records for which security clearance is to be determined. The record memory is also used to maintain (separate) queues of requests from the MM and DBCCP.

We now describe the logic of the MC as it processes a security clearance



Note: The same arrangement can be used for processing the contents of the three paths by a single processor P1. P2 and P3 merely maintain the interconnection between adjacent paths.

Figure 58. Multiple Paths (3) in Tandem Shown Here During Loading

request. First, the command identifier is extracted from the request. This identifier is used to access the command status table (CST) and retrieve the user ID of the user who issued the command. This ID is then used to access the user information table (UIT) to obtain the address of the file sanctions in the security information table. The query associated with a file sanction is then loaded into the query memory. The keywords of the records in the record memory associated with the request under processing, are loaded into the comparison memory as follows: the "connect" signal C_i of all the PEs (see Figure 57) except the first PE (see Figure 58) are raised high. This forces all the PEs except the first to allow data to be exchanged between adjacent paths. As the bit stream of a keyword set reaches its assigned path, the connect signal of the corresponding PE is lowered, thus "trapping" the keyword set in the data path. If there are n paths in a chip, and if we assume that all chips in the comparison memory are loaded simultaneously, the time taken to load the comparison memory is n path circulation periods. After the loading of the comparison is completed, the MC proceeds to systematically search for keywords in the data path which satisfy the keyword predicates of the query conjunct in the query memory. The MC also maintains the correspondence between the sets of keywords in the data paths and the records in the record memory in which the keyword sets occur. At the end of one search operation, the records which have keywords satisfying the file sanction query in the query memory are identified. The access descriptor associated with the file sanction is looked up to check if the access requested on the records is granted. If it is not, then the records which satisfied the query are deleted from the record memory.

The above process is repeated for each of the file sanctions in the user's database capability. The records, which survive the deletions at the end of each of the search operation, are then sent to the DBCCP or the MM. These are the records for which the requested access is granted by the database capability of the user.

4.2.2 Sort Module (STM)

Records which are cleared for security by the SEM, and are to be sent to the PES (via the DBCCP) are sorted by the sort module (STM) if the sort option has been specified by the user. The STM uses hardware very similar to the hardware used by the SEM. The STM consists of three components - a sort memory, a sort memory controller and a record memory. The sort memory is implemented by a set of sequential access memories called data paths and by a corresponding set of processors. Each processor can receive inputs from two adjacent data paths. The sort method used is the odd-even transposition sort described in [14]. Each data

path is loaded by the sort memory controller with a value of the attribute specified in the sort option and which occurs in a record to be sent to the PES. The data paths are designed to be long enough to hold values up to 31 bytes long (recall from section 2 that alphanumeric values can be no longer than 31 bytes). The sort memory controller also loads into each data path a pointer to the record in which the keyword, with the value in the path, occurs. The maximum time to sort is given by n/2 path circulation periods, where n is the number of paths in the sort memory. The record memory contains the records while the sorting is in progress.

As in the case of the SEM, the processing element (PE) in the sort memory is an important component. We have shown a block diagram of a PE in the sort memory in Figure 59a. The logic of a PE is as follows. A PE can function in three modes - the load mode, the sort mode, and the read mode. During the load mode all PEs except the PE_o (see Figure 59b) connect L_i to L'_{i-1} and L_{i-1} to L'_i. PE_o connects data input line D_{in} to L_o. All the data paths in the sort memory can be loaded in n circulation periods. After loading is completed, the sort memory controller raises the sort signal S_E for half a circulation period. This begins the sort mode. During the next half a circulation period, the sort signal S_o is raised. The sort signal S_E initiates all even numbered processors to compare adjacent path contents, while the sort signal S_o initiates all odd numbered processors to compare adjacent path contents. When the contents of adjacent paths are different and they meet the sort criterion (ascending or descending order), then the contents are interchanged between paths by the processor. For example, if the sort order specifies ascending order, and if $L_{i-1} > L_i$, L_i is connected to L'_{i-1} and L_{i-1} is connected to L'_i. The comparison is always bit serial, and, therefore floating point numbers should be normalized for correct sorting. The shifting of data paths past the processing element is synchronous. If during a circulation period, no exchanges take place, the sort memory controller recognizes this fact by sensing the signal marked x. This signal is propagated from one PE to the next; at each PE the incoming signal is ORed with an exchange signal locally generated.

After the sorting is completed, the sort memory enters the read mode if the signal R is raised by the memory controller. During the read mode, all PEs except PE_o connects L_i to L'_{i-1} and L_{i-1} to L'_i. PE_o connects L'_o to D_{out}. The sort memory controller retrieves the records from the records memory in the order in which pointers are retrieved from the sort memory. The retrieved records are sent to the DBCCP and palced in the data response memory (see Figure 41).

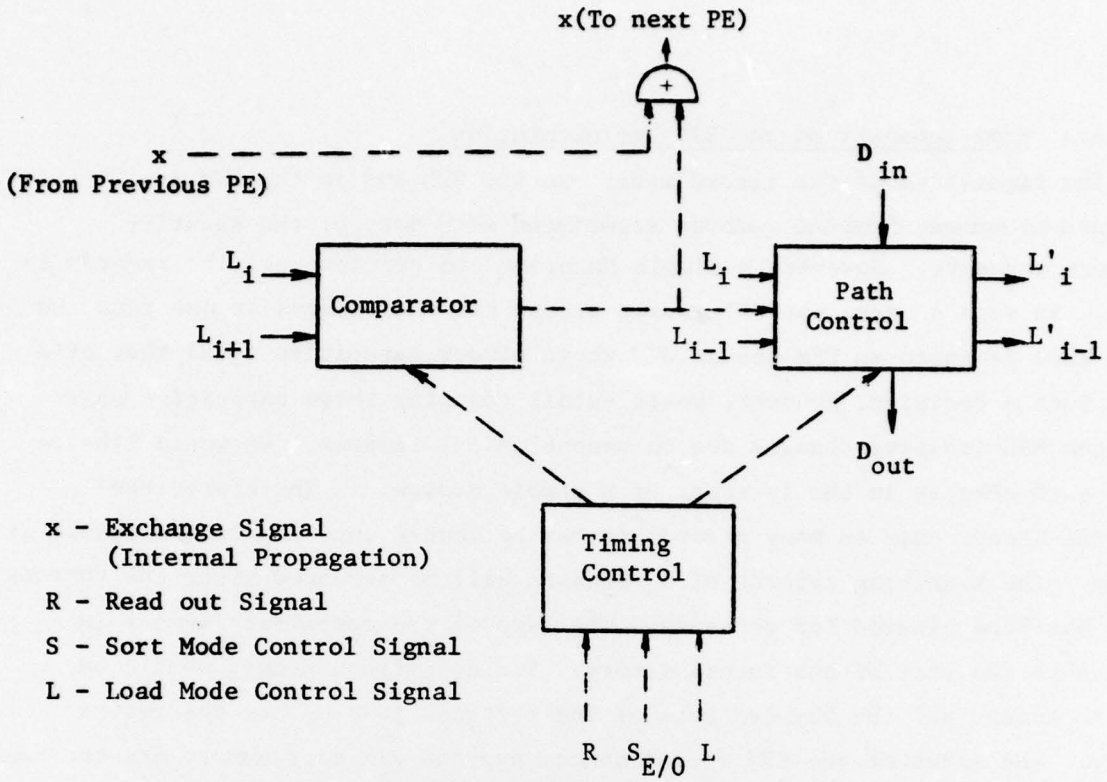


Figure 59a. Block Diagram of a PE in the Sort Memory

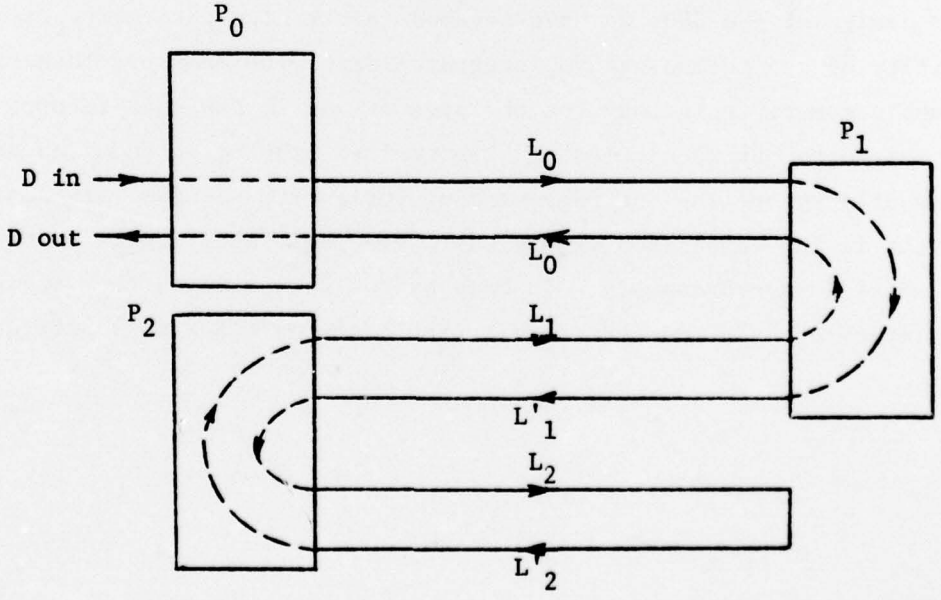


Figure 59b. Three - Path Sort System [All paths are moved synchronously]

4.3 Some Comments on the SFP Implementation

The capacities of the record memory in the SEM and in the STM are designed to accommodate the records associated with most of the security and sort requests. However, a single MM order can retrieve all the records in a MAU. In such a case, the SFP cannot accept all the records at one time. We could have designed an SEM and an STM whose memory capacities equal that of a MAU. Such a decision, however, would entail changing these capacities as and when the MAU capacity changes due to technological reasons. We would like to avoid such changes in the interest of a stable design. Therefore, the SFP will accept only as many records as can be loaded into the record memory at a time. The remaining records of a request, will be accepted after the current batch has been cleared for security. The size of the comparison memory is related to the size of the record memory. The comparison memory should be able to accept all the keyword sets of the records loaded into the record memory. The sizes of the STM record memory and the STM sort memory are the same as their counterparts in the SEM. This is because it is possible that all the records in the SEM are cleared for security. In this case, they will be sent to the STM which should be in a position to accept them so as to force the SEM to process the next request.

In our design of the SFP, we have assumed fairly high data shift rate (10MBS) and the ability of the technology to integrate logic with memory. These two rule out bubble memory technology for the present and in the near future. CCDs are better suited to our requirements. The refresh problem in CCDs can be handled elegantly by merging the regenerating logic with the PEs. In addition, the data paths in the comparison memory must have individual clocks to enable them to be shifted asynchronously. In case of the sort memory, the data paths are shifted synchronously and therefore a single common clock will suffice.

5. CONCLUDING REMARKS

In the preceding sections, we have endeavored to present the design details of three components of the DBC: the DBCCP, the MM and the SFP. The MM and the SFP formed the data loop as shown in Figure 1. The DBCCP played essentially the role of an interface between the environment represented by the PES on the one hand and the structure and data loops on the other. As in an earlier report [7], one of the main objectives in writing this report was to demonstrate the feasibility of the DBC as envisioned in [2].

An important aspect of the design is the level of modularity achieved within each of the components. The DBCCP had three processors, and a number of table memories; the MM had a set of track processing elements communicating with a controller whose logic was shared by two subprocessors, while each of the track processors was an assemblage of two micro-sequencers and small sequential memories; and the SFP had two intelligent memories in tandem. The identification of these subcomponents enabled us to adopt a step-at-a-time approach to the DBC design. Of course, such a design has the usual salutory effects on reliability and fault isolation.

As a result of the current effort, two avenues of further work are suggested herein. First, a cost-performance evaluation and functional verification of the proposed design must be undertaken before a prototype can be built. Such a study is now in progress. Second, from a user point of view, the capabilities of the machine to support existing data models must be demonstrated. Since the DBC directly supports (i.e., with hardware data structures) an enhanced version of the attribute based model [2], software interfaces to map other data models into the DBC-supported model must be constructed. This area is also being investigated. We believe that such interfaces are easier to build and more efficient than contemporary databases built on the conventional hardware.

REFERENCES

1. Hsiao, D. K., Systems Programming - Concepts of Operating and Database Systems, Addison-Wesley, Reading, MA, 1975, Ch. 6.
2. Baum, R. I., Hsiao, D. K. and Kannan, K., "The Architecture of a Database Computer, Part I: Concepts and Capabilities", The Ohio State University, Tech. Rep. No. OSU-CISRC-TR-76-1, (September 1976).
3. McCauley, E. J. III, "A Model for Data Secure Systems", Ph.D Dissertation, Department of Computer and Information Science, The Ohio State University Tech. Rep. No. OSU-CISRC-TR-75-2, (1975).
4. Hsiao, D. K. and Harary, F., "A Formal System for Information Retrieval from Files", Communications of the ACM, 13, 2, (February 1970), 67-73; Corrigenda, CACM (March 1970).
5. Wong, E. and Chiang, T. C., "Canonical Structure in Attribute Based File Organization", Communications of the ACM, 14, 9, (September 1971), 593-597.
6. Knuth, D. E., The Art of Computer Programming, Vol. I - Fundamental Algorithms, Addison-Wesley, Reading, MA, 1973.
7. Fagin, R., "Some Comments on the Architecture of a Database Computer", Personal Communication to D. K. Hsiao, (December 1976).
8. Hsiao, D. K. and Kannan, K., "The Architecture of a Database Computer, Part II: The Design of Structure Memory and its Related Processors", The Ohio State University Tech. Rep. No. OSU-CISRC-TR-76-2, (October 1976).
9. Hoagland, A. S., "Magnetic Recording Storage", IEEE Transactions on Computers, C-25, 12, (December 1976), 1283-1289.
10. Bremer, J. W., "Hardware Technology in the Year 2001", Computer, 9, 12, (December 1976), 31-36.
11. Ozkarahan, E. A., et al., "RAP: A Rotating Associative Processor for Database Management", Proceedings of the AFIPS National Computer Conference, 44, (1975), 379-387.
12. Edelberg, M. and Schissler, L. R., "Intelligent Memory", Proceedings of AFIPS National Computer Conference, 45, (1976), 393-400.
13. Stone, H. S., "The Organization of Electronic Cyclic Memories", Computer, 9, 3, (March 1976), 45-50.
14. Knuth, D. E., The Art of Computer Programming, Volume III: Sorting and Searching, Addison-Wesley, Reading, MA, 1973.